MASTER EN SCIENCES INFORMATIQUES

L'étude des triggers et des stored procedures dans les bases de données actives

Dawance, Jean-Pol

Award date:
1999

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX, NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B-5000 NAMUR (BELGIUM)

L'étude des triggers et des stored procedures dans les bases de données actives

Dawance Jean-Pol

Mémoire présenté en vue de l'obtention du grade de

Licencié en Informatique

Année Académique 1998 - 1999.

AVANT PROPOS

Fruit du travail d'un étudiant, un mémoire est également le reflet de la collaboration de nombreuses personnes. C'est pourquoi je tiens à adresser mes plus sincères remerciements à Monsieur Jean-Luc Hainaut, mon promoteur pour la disponibilité et l'implication dont il a fait preuve tout au long de cette recherche.

Que Messieurs Koen Bertels et Georges Schepens, professeurs à la Faculté des Sciences Économiques Sociales et de Gestion, ainsi que Monsieur Marc Vernaillen, Account Manager Government chez Oracle Belgique, soient également assurés de ma profonde reconnaissance pour leurs nombreux conseils et renseignements qu'ils m'ont communiqués.

La rédaction d'un travail est une chose, ses relectures en sont une autre. C'est pourquoi je remercie particulièrement Cédric, Catherine, Luc, Christine, Jean-François, Emmanuel, Frédéric et Michel pour leur dévouement et leur encouragement.

Un merci particulier doit être également adressé à Messieurs Charles Jaumotte, Administrateur Délégué des FUNDP et Paul Reding, ancien Doyen de la Faculté des Sciences Economiques Sociales et de Gestion pour la confiance qu'ils m'ont témoignée.

Enfin, je tiens à exprimer toute ma gratitude envers mes parents, collègues, professeurs, proches et ami(e)s qui m'ont soutenu durant l'élaboration de ce travail.

Table des matières

Table des Matières

Introduction

Introduction

Les bases de données actives commencent à jouer un rôle prépondérant dans les applications commerciales de gestion de données. Depuis l'apparition des systèmes client/serveur, de nombreuses applications ont été construites par des petits groupes de développeurs souvent autonomes. Ces applications offrent généralement une vision assez étroite et peu précise de l'ensemble de l'information de l'entreprise. Dès lors, le système de gestion du flux de l'information est relativement vulnérable aux violations de contraintes d'intégrité.

Les bases actives permettent d'assurer un contrôle continu sur la base de données. En détectant certaines occurrences d'événements, elles peuvent, sans aucune intervention de la part de l'utilisateur, déclencher des actions de contrôle d'intégrité, de vérification d'accès, de réparations de contraintes et d'audit. Ces actions peuvent être définies à l'intérieur ou à l'extérieur de la base de données. Elles offrent une extension très importante par rapport à ces prédécesseurs dits passifs. En effet, l'introduction des règles actives permet d'apporter une nouvelle dynamique dans les systèmes de gestion de bases de données. Ces règles actives implémentent le modèle Evénement-Condition-Action. Elles apportent une nouvelle dimension au développement des applications de bases de données et permettent d'étoffer leurs domaines d'application.

Actuellement, les systèmes de gestion de bases de données actives commencent à être reconnus comme une technique stratégique à intégrer dans la gestion des bases de données. Cette reconnaissance émerge, à la fois de la communauté scientifique qui étudie, évalue et développe, depuis une dizaine années, de nouveaux langages et leurs architectures associées mais également des firmes commerciales qui ont rapidement intégré ces techniques (triggers et procédures stockées) dans leurs produits par la création de nouveaux langages (SQL3, PL/SQL, TRANSACT-SQL).

Cette nouvelle dimension permet d'apporter de nouvelles fonctionnalités aux applications existantes mais également d'aborder des domaines encore peu explorés (surveillance de réseaux, contrôle des systèmes d'information,...). Cependant, par la méconnaissance ainsi que par leur complexité de modélisation, ces techniques sont peu connues, parfois mal comprises et trop peu mises en pratique.

Notre travail s'intègre dans cette problématique. Il a pour objectif de fournir une introduction au concept des bases de données actives et des techniques permettant d'introduire ces processus actifs dans les systèmes de gestion de bases de données.

Notre travail débute par une définition des bases de données actives à travers la mise en évidence de leurs caractéristiques et par la présentation du modèle ECA (Evénement-Condition-Action). En effet, ce modèle permet d'implémenter les règles actives selon une logique : lorsqu'un événement se produit et que les conditions de déclenchement sont

satisfaites, une action est exécutée. Nous analysons alors les différents composants de ce modèle et dégageons les possibilités d'implémentation.

Nous classifions ensuite les différents systèmes actifs en fonction de deux critères : d'une part, le rôle qu'il joue dans le système d'information et d'autre part, son degré d'intégration dans celui-ci.

Notre second chapitre aborde l'utilité ainsi que les applications des règles ECA. Leurs domaines d'application sont relativement vastes et diversifiés mais leurs fonctions s'intègrent principalement dans le contrôle de l'intégrité référentielle (vérification de contraintes et correction en cas de violation), la gestion des transactions avancées et des vues, et l'intégration de données dans le cas de bases distribuées. Nous développons leurs fonctionnalités et leur intégration dans les systèmes de gestion de données.

Après cette analyse théorique des systèmes actifs, nous abordons la mise en pratique de ces concepts à travers l'analyse des triggers et des procédures stockées, et leur implémentation dans le langage PL/SQL.

Le langage PL/SQL est un langage créé par Oracle pour implémenter, entre autres, les règles actives. Nous présentons les principales caractéristiques de la dernière version de ce langage (version 8.0). En complément, une brève description de sa syntaxe, des structures et des types de données qu'il peut manipuler, est également explicitée.

Le chapitre suivant est consacré aux procédures stockées. Après les avoir définies, nous développons les trois types de procédures stockées : les procédures, les fonctions et les packages. Nous exposons leurs méthodes d'implémentation en PL/SQL et leurs propriétés. Conjointement, nous dégageons les avantages de l'utilisation de ces éléments au niveau de la sécurité, de la performance et de la productivité des bases de données.

Les triggers sont, elles, des procédures stockées particulières permettant d'implémenter les règles ECA. Nous en étudions la théorie avant d'en formaliser l'implémentation en PL/SQL de même que leurs utilités au niveau de l'intégrité référentielle, de la sécurité, du contrôle des accès et des audits possibles. Une bonne compréhension des triggers, conjuguée à des conventions d'écriture, permet de mieux les analyser et de faciliter leur création.

Malgré leur potentiel, les triggers connaissent certaines limites que nous tâchons de dégager. En effet, les opérations exécutées par les triggers (row-level) sont restreintes. Nous présentons ces quelques limitations et plus globalement, nous dégageons les faiblesses actuellement recensées des techniques de modélisation des triggers.

Nous nous intéressons également à la problématique de la modélisation des contraintes sous la forme de contraintes déclaratives ou procédurales (avec les procédures stockées ou de triggers). Une bonne connaissance de leur mise en pratique permet de simplifier certaines modélisations.

Notre partie pratique aborde, à travers un exemple simplifié, l'implémentation de règles stratégiques sous la forme de triggers et de procédures stockées. Nous développons ainsi une méthodologie d'implémentation des triggers.

Par cette méthode de développement, il est possible, de manière systématique, de modéliser des règles stratégiques et d'éviter les problèmes associés aux tables mutantes. Nous évoquons ce problème de tables mutantes, ses conditions de survenance et analysons une méthode de résolution (table temporaire PL/SQL).

Enfin, notre dernier chapitre expose quelques subtilités d'écriture permettant l'implémentation de triggers récursifs et de triggers associés à des opérations de suppressions en cascade (DELETE CASCADE).

Notons encore que ce travail ne constitue qu'une introduction à la problématique globale de l'utilisation des triggers et des procédures stockées dans les bases de données actives car à l'heure actuelle, les progrès en cette matière ne font qu'apporter de nouveaux éléments qui étayent davantage l'intérêt de ces techniques. Malgré leur faible utilisation, les années futures prouveront, grâce à une meilleure compréhension des concepts associés et à des techniques plus efficaces, que les triggers constituent une dimension stratégique des bases de données. Notre travail tente de prouver cette tendance.

Chapitre 1:

Les bases de données actives

1.1 Introduction

1.2 Intégration des règles stratégiques de gestion par le modèle ECA

1.2.1 Événements (Events)

1.2.2 Conditions

1.2.3 Actions

1.2.4 Remarques sur le modèle ECA

1.3 Caractéristiques des bases de données actives

1.3.1 Définition d'un SGBDA

1.3.2 Résumé des caractéristiques d'un SGBDA

1.4 La structure d'un SGBDA implémentant des règles actives

1.5 Classification des SGBDA

1.5.1 Un SGBDA "Monitoring" dans un SI Homogène

1.5.2 Un SGBDA "Control" dans un SI Homogène

1.5.3 Un SGBDA "Control" dans un SI Hétérogène

1.5.4 Tableau récapitulatif

1.6 Résumé

Chapitre 1 :

Les bases de données actives

1.1 Introduction

Les systèmes de gestion de base de données sont actuellement au cœur des technologies de l'information. Ils fournissent des mécanismes fiables, efficaces et pertinents en matière de sauvegarde et de gestion de gros volumes d'informations dans des environnements multi-utilisateurs et distribués. Depuis quelques années, les recherches en matière de base de données et leur pratique dans la vie courante ont abouti à de nouvelles sémantiques pouvant être supportées par les systèmes de base de données. Les bases de données temporelles, spatiales voire multimédia sont des exemples de cette tendance de développement. Les bases de données actives s'insèrent dans ce courant où la sémantique supportée est le reflet du comportement du domaine. Ce comportement est basé sur des événements qui peuvent être vus comme des situations spécifiques nécessitant une ou plusieurs réactions. Ce comportement par lequel des actions sont effectuées en réponse à la survenance d'un événement particulier est appelé le comportement réactif ou *reactive behaviour*.

Un système de gestion de base de données active (SGBDA) est un système qui détecte certaines situations (supposées intéressantes, voire anormales), évalue les conditions par lesquelles cette situation s'est produite et, si une certaine condition est remplie (expression booléenne évaluée à *true*), il exécute une action d'une manière opportune. Ces règles sont définies comme des règles ECA (Evénement-Condition-Action). Elles permettent d'introduire au sein de la base de données des règles stratégiques de gestion (sécurité, contrainte d'intégrité, performance..). Le principe du modèle ECA peut se définir comme suit : lorsque un événement particulier se produit et qu'une condition de réalisation est vérifiée, une action spécifique est exécutée. Nous décrirons ce principe plus en détails au cours de notre travail.

Les SGBDA peuvent permettre d'assurer plusieurs fonctions stratégiques telles que la gestion des contraintes, le contrôle des accès et la gestion des longues transactions. Ceux-ci sont en contraste avec les systèmes classiques de bases de données passives qui exécutent uniquement des requêtes ou des transactions explicitement définies par un utilisateur ou un programme d'application.

Prenons l'exemple suivant : lorsque la quantité en stock d'un certain produit se situe en deçà d'un seuil fixé, une action de commande de ce produit est initiée. Cette situation peut être implémentée de deux manières dans les bases de données passives. Cependant, quelle que soit la manière choisie, les résultats sont peu satisfaisants.
La première solution suggère que chaque programme qui effectue une mise à jour dans la table *inventaire* (table de gestion des stocks) doit vérifier la contrainte de limitation et au besoin, invoquer l'opération de renouvellement de commande. Le principal désavantage de cette approche est que la modification ou la suppression de la contrainte exige de retrouver et de modifier le code associé dans chacune des applications utilisant la règle.
Une deuxième manière est d'ajouter un programme spécial dont le but est d'effectuer une requête sur la base de données afin de vérifier les contraintes spécifiées et auquel chaque application doit faire appel. Cependant, effectuer trop souvent ces vérifications peut nuire à l'efficacité. En revanche, exécuter ces vérifications de manière moins fréquente peut conduire

à des réponses retardées par rapport à des situations critiques. Ces deux situations sont donc peu convaincantes.

Les bases de données actives permettent d'introduire cette vérification d'une manière beaucoup plus efficace en spécifiant des règles qui sont définies et stockées directement dans la base de données. L'avantage de cette technique est que l'ensemble des programmes peut partager ces règles et que le système peut optimiser leur implémentation.
Les recherches en matière de bases de données actives se sont fortement développées et abordent des domaines d'applications très diversifiés qui étaient difficilement analysables par les systèmes traditionnels de gestion de bases de données. Ces domaines sont, entre autres, la gestion des réseaux, le contrôle du trafic aérien, la gestion boursière...

Nous allons, d'une part, décrire le modèle ECA et, d'autre part, énoncer les caractéristiques que doit posséder une base de données pour être déclarée active. Trois articles forment la base de références de notre réflexion :

1. G. Knolmayer, H. Herbst et M. Schilesinger, *"Enforcing business Rules by applications of triggers Concepts"*, Information Conference, Swiss National Science Foundation,1994, p. 24-30.
2. K. Dittrich, S. Gatziu et A. Geppert, *"The active database management system manisfesto : a rulebase of ADBMS features"*, ACT-NET Consortium, 1997.
3. S. Chakravarthy : "special issue on active databases", Bulletin of the TC on data Engineering 15, 1992, 1-4.

1.2 Intégration des règles stratégiques de gestion par le modèle ECA

Les règles de gestion peuvent être représentées suivant la structure de règles des bases de données actives : c'est-à-dire à travers les trois composants du modèle Evénement-Condition-Action que nous allons décrire en détails.

1.2.1 Événements (Events)

Les événements peuvent être élémentaires ou complexes.

Les événements élémentaires correspondent à des occurrences simples qui ne peuvent être décomposées. En effet, il est possible de distinguer :

Les événements associés aux données : ils peuvent être identifiés par une action portant sur un objet "donnée" (*data*) (table, colonne, procédure, fonction...) de la base de données. Cet objet est stocké dans la base de données.

Les événements temporels : ce sont les règles stratégiques qui se déclenchent à des moments bien précis.

*Les événements définis par l'utilisateur : i*ls couvrent les situations se produisant dans d'autres circonstances que les deux points précédemment cités et qui sont explicitement définies par une application ou un utilisateur.

Les événements complexes correspondent à des combinaisons d'événements élémentaires ou complexes possédant des caractéristiques complémentaires. Il est possible de distinguer plusieurs types d'événements complexes :

Événements "Booléens"

- Disjonction ou conjonction de deux ou plusieurs événements comme par exemple, la conjonction de ces deux événements :
  [variation de plus de 2% d'un cours de bourse] et [produit disponible sur le marché]

- Sélection d'événements parmi une liste : par exemple, la règle de l'événement complexe est la survenance de trois événements parmi une liste donnée ($Evénement_1$, $Evénement_2$,....., $Evénement_{n-1}$, $Evénement_n$). Ce principe peut être également exprimé par une combinaison de conditions booléennes mais cette méthode permet une description de la sémantique d'une manière plus lisible.

- Séquence d'événements ($Evénement_1$; $Evénement_2$; ....., $Evénement_{n-1}$; $Evénement_n$). Cette méthode correspond à la conjonction d'événements successifs qui doivent se réaliser dans un délai imparti.

Événements "Temporels"

- Événements à intervalle : ce sont des événements qui se produisent entre deux situations particulières. Par exemple, [achat d'un produit] entre [une offre proposée] et [quinze jours après offre].

- Événements périodiques : ce sont des événements qui se produisent périodiquement. Par exemple, la consolidation des comptes est effectuée tous les jours à 00h00; toutes les cent transactions, un contrôle est effectué.

- Événements retardés : ce sont des situations qui se réalisent dans un délai bien précis après la survenance d'un autre événement (quinze jours après l'envoi d'un courrier, une proposition d'offre est effectuée).

1.2.2 Conditions

Tout comme les événements, elles peuvent être classées en conditions élémentaires ou complexes.

Une condition élémentaire est simplement un prédicat ou une requête portant sur les données d'une base de données.

- une comparaison entre un objet et une valeur : val_min_compte > 1000

- une comparaison entre la valeur de deux objets :
  valeur_min_compte > compte_min

- une opération particulière sur les données ($\forall$ clients, $\exists$ compte..)

Une condition complexe regroupe un ensemble de conditions simples ou complexes qui sont combinées entre elles par un opérateur booléen (et, ou, not, =,..).

1.2.3 Actions

Les actions résultent de la double survenance de l'événement et de la condition de réalisation. De cette manière, il est logique d'associer, de manière similaire, les types d'actions et les événements associés.

Événements associés aux données : ces actions peuvent effectuer des opérations d'insertion, de récupération, de mise à jour et de suppression de l'objet "donnée" (*data*) associé.

Message d'actions : ces actions contiennent un message vers un utilisateur; ce message peut provenir d'une application ou d'un autre utilisateur.

Actions d'un utilisateur : elles sont effectuées par l'utilisateur du système et outrepassent les messages d'actions.

1.2.4 Remarques sur le modèle ECA

- Il est parfois possible que la condition ne soit pas spécifiée. De cette manière, le modèle devient Evénement - Condition. En effet, si le type d'événement est obligatoire, la partie condition peut être facultative puisque celle-ci peut, à la convenance du créateur de la règle, être transférée dans la code de l'action.

- Il se peut que l'événement soit implicite; la règle se transforme en une règle CA. Dans certains cas, il est préférable au compilateur ou au SGBDA, lui-même, de générer la définition de l'événement; l'événement est, par conséquent, défini de manière implicite. Si l'utilisateur spécifie les conditions et les actions, c'est le SGBDA qui détermine l'événement automatiquement. Imaginons un mécanisme implémentant le principe de cohérence dans une base de données. Les contraintes et les règles de remise en état étant définies, le système utilise, de manière interne, les événements observés par les modifications sur la base pour déterminer les contraintes devant être respectées. Cependant, ce système doit offrir la possibilité à un utilisateur de définir lui-même des événements. En cas de non spécification externe, le système peut générer implicitement les événements.

Nous verrons par la suite comment ces règles peuvent être implémentées, appliquées et utilisées dans la vie courante à travers les bases de données actives.

1.3 Caractéristiques des bases de données actives

1.3.1 Définition d'un SGBDA

1. *Un système de gestion de bases de donnée active (*SGBDA*) est un système de gestion de base de données (*SGBD*).*

Un SGBD est un système de gestion d'informations qui peut manipuler une multitude de données stockées dans la base de données. Un SGBD dépasse le simple cadre de la gestion de simples variables ou de fichiers de données. Dans le système, les données sont définies avec

les relations entre les données qui sont stockées au sein des tables et qui peuvent être gérées par des requêtes ou des vues.
Manipulant des données de différents types ainsi que des "méta-données" afin de définir le schéma de la base, le SGBD garantit la pérennité des données en s'assurant qu'aucune d'entre elles ne soit perdue en cas de défaillance du système. La gestion des transactions est une des principales fonctionnalités d'un SGDB. Les transactions fournissent un mécanisme pour organiser et synchroniser les opérations sur la base. De cette manière, plusieurs utilisateurs ou applications peuvent utiliser des transactions - permettant de spécifier une séquence d'opérations à réaliser - et cela, sans tenir compte des interactions avec les autres utilisateurs ou applications.
Tous les concepts proposés par un système passif sont inclus dans un système actif. Citons, par exemple, le langage des requêtes, les accès multi-utilisateurs, le système de récupération...
En résumé, si un utilisateur ignore toutes les fonctionnalités actives du système, un SGDBA peut être utilisé de manière équivalente à un SGBD passif.

2. *Un SGBDA applique le modèle ECA (Événement, Condition, Action).*

Le SGBDA étend un SGDB passif par l'introduction de comportements réactifs qui sont définis par les utilisateurs. La manière de spécifier les règles de comportement du système est modélisée par les règles ECA telles que nous les avons définies précédemment. L'utilisation de ce modèle implique trois principes fondamentaux :

- *Le SGBDA doit fournir des moyens pour définir les types d'événements.*

  Les types d'événements décrivent les situations dans lesquelles le système doit réagir. En général, il faut distinguer les "before events" et les "after events". Dans le cadre des opérations de bases de données, les "before events" sont directement signalés avant que l'opération ne soit réellement exécutée. Les "after events", quant à eux, sont signalés directement à la fin de l'exécution de l'opération. Ces deux types d'événements peuvent incorporer les opérations DML[1] et les instructions de transactions. Il est généralement possible d'associer des paramètres à la survenance d'un événement tels que le nom de l'utilisateur qui est à l'origine de la transaction.

- *Le SGBDA doit fournir des moyens pour définir les conditions.*

  La condition exprime un état dans lequel doit être la base de données afin d'exécuter l'action associée lorsqu'un événement est constaté. Elle exprime, en quelque sorte, ce qui doit être vérifié pour appliquer la troisième étape du modèle. Cette condition est vérifiée après que le type d'événement soit constaté (la règle se déclenche et la condition est examinée). Cette condition peut être un prédicat sur l'état de la base de données (WHERE) ou une requête sur la base de données. La condition est satisfaite si le résultat est évalué à vrai ou non-vide.

[1] Les commande DML (Data Manipulation Language) manipulent de l'information au sein d'une base de données à travers les commandes de lecture (SELECT) , d'insertion (INSERT), de mise à jour (UPDATE) et de suppression (DELETE). En outre, ce sont les commandes DDL (Data Definition Language) qui permettent de créer les objets de la base de données (tables, contraintes, procédures stockées et triggers).

- *Le SGBDA doit fournir des moyens pour définir les actions.*

  L'action permet de spécifier la réaction associée à la survenance de l'événement. Elle est directement exécutée lorsque la règle est déclenchée (survenance de l'événement) et que la condition est vérifiée. Classiquement, les actions peuvent effectuer de multiples opérations : modification des données, récupération de données, opérations de transaction (commit, rollback), appel à des procédures stockées ou externes et utilisation des commandes DML.

3. *Un SGBDA doit supporter la gestion des règles ECA ainsi que leur évolution.*

- *Le SGBDA doit supporter la gestion des règles.*

  L'ensemble des règles est défini à un moment donné en un lieu déterminé. Ces règles doivent être gérées par le système. En résumé, la définition des règles ECA doit être intégrée dans l'information du système et être stockée dans la base de donnée. En effet, toutes les informations relatives à ces règles doivent être définies de manière à ce que l'on puisse déterminer les règles existantes et voir comment elles ont été spécifiées. Tous les utilisateurs ou toutes les applications peuvent consulter ces renseignements via une sorte de catalogue regroupant les règles définies.

- *Le SGBDA doit supporter l'évolution des règles.*

  Les règles définies doivent subir au fur et à mesure de leur utilisation, certaines adaptations. En effet, un SGBDA doit permettre l'insertion de nouvelles règles ECA, ainsi que la suppression de règles, voire l'adaptation de certains événements, conditions ou actions.

- *Le SGBDA doit supporter l'activation et le blocage de règles.*

  Une règle peut être, à certains moments, active et désactivée à d'autres moments. Désactiver une règle ne signifie pas que la règle soit supprimée de la base mais simplement que si l'événement est constaté, la règle n'est pas appliquée, elle est simplement gelée. Une règle est active si, lorsque l'événement est constaté et que la condition est vérifiée, l'action est exécutée.

4. *Le SGBDA doit posséder un modèle d'exécution.*

- *Le SGBDA doit pouvoir détecter les occurrences d'événements.*

  Un SGBDA détecte idéalement toutes les survenances d'événements quels qu'ils soient. Cependant, cette situation est rare, pour ne pas dire utopique. En effet, certaines survenances doivent être explicitement signalées par l'utilisateur ou par l'application elle-même. Dans les cas extrêmes, les programmeurs sont responsables de signaler l'ensemble des types d'événements. Cette dernière situation correspondrait à un DBMS passif où les utilisateurs ont les privilèges d'indiquer les événements explicitement.

- *Le SGBDA doit supporter un mode de liaison.*

  Les événements sont généralement associés aux données de la base. Le type de liaison détermine la granularité des événements associée à la règle. Différents types de liaison existent :

  - *"Instance Oriented"* : ce type de liaison signifie que les objets spécifiques (simples ou composés) pour lesquels un événement se produit, peuvent être référencés par les conditions ou les actions. De cette manière, si un même événement est associé à plusieurs enregistrements, alors la règle associée sera exécutée individuellement pour chacun des enregistrements.

  - *"Set-Oriented"* : En revanche, pour ce type de liaison, la survenance de l'événement est associée à un ensemble d'enregistrements pour lequel l'événement est apparu. Dans ce cas, la règle associée est exécutée une et une seule fois pour tous les enregistrements liés à l'événement déclencheur. Les conditions et les actions peuvent uniquement faire référence à cet ensemble.

  Cette distinction est implémentée en Oracle par les triggers "Row Level" et "Statement Level".

- *Un SGBDA doit être capable d'évaluer les conditions.*

  Quel que soit l'événement déclencheur, le système doit pouvoir évaluer les conditions. Dans la pratique, il est nécessaire que les événements fournissent de l'information aux conditions puisque, si un événement s'est produit pour un objet particulier ou un ensemble d'objets, il doit être possible de faire référence à cette information dans la condition.

- *Un SGBDA doit être capable d'exécuter les actions.*

  Une fois, l'événement détecté et la condition vérifiée, le système doit pouvoir exécuter l'action de la règle associée. De même que le point précédent, il doit être possible de faire référence à de l'information provenant des parties Événement ou Condition.

5. *Le SGBDA doit pouvoir proposer différents modes d'association (coupling modes).*

Les modes d'association sont définis par la relation qui est spécifiée entre d'une part, l'évaluation des conditions de la règle et l'exécution de l'action associée et d'autre part, l'événement déclencheur et la transaction dans laquelle l'événement s'est produit.
Citons les différentes modes possibles :

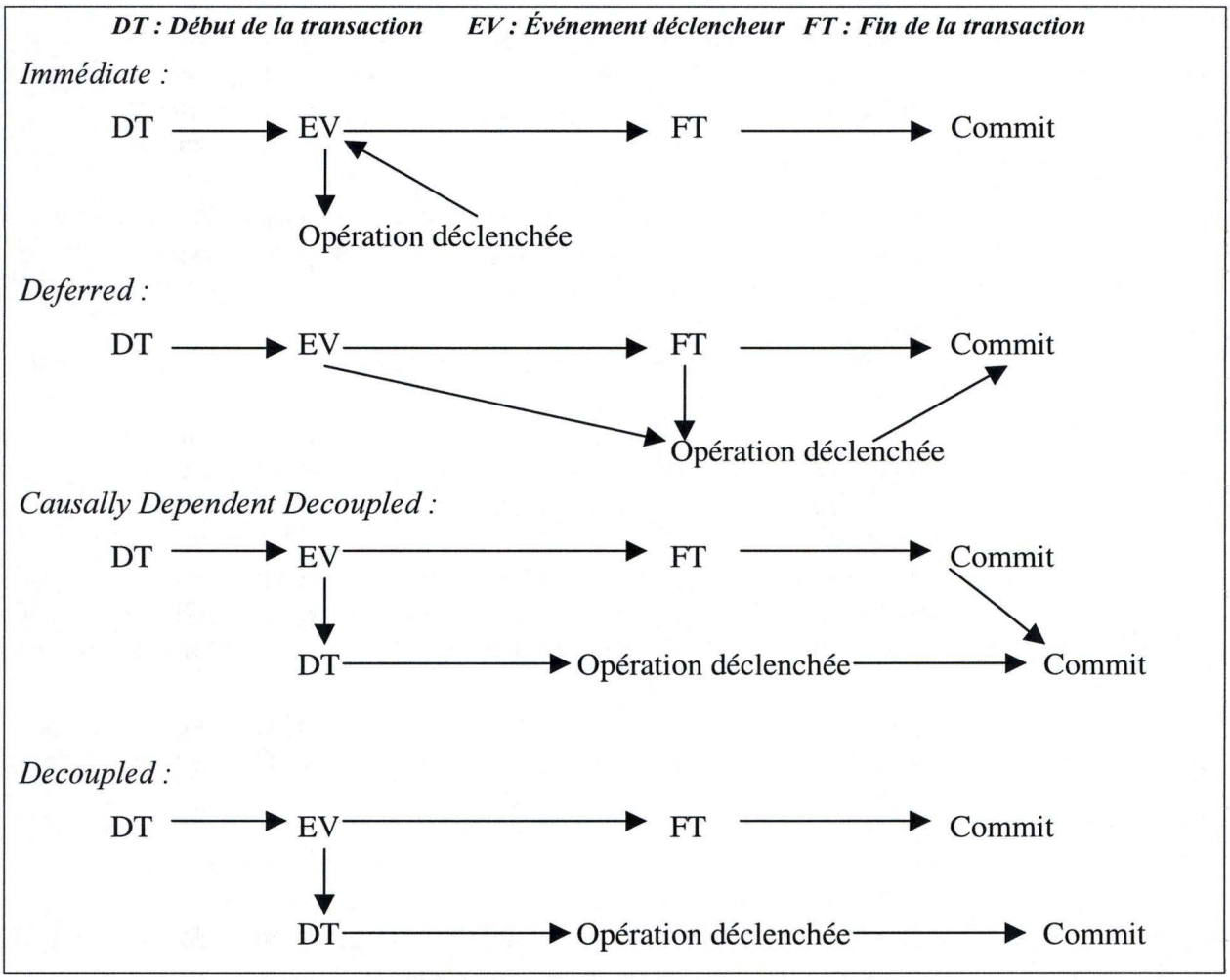
- *Immediate Rule Processing* : les conditions de la règle sont évaluées et les actions sont exécutées immédiatement après l'occurrence de l'événement.

- *Deferred Rule Processing* : le traitement des règles est retardé jusqu'au moment où la transaction doit être confirmée (commande COMMIT).

- *Causally Dependent Decoupled Rule Processing* : l'exécution d'une action déclenchée est réalisée par l'intermédiaire d'une sous-transaction qui attend la confirmation de la transaction principale pour effectuer son opération d'approbation.

- *Decoupled Rule Processing* : la sous-transaction est réalisée complètement indépendamment de la transaction principale. Elle est, dès lors, séparée de la transaction principale et peut confirmer ces changements sans se préoccuper des résultats de la transaction principale.

Schématisons ces 4 types de traitements afin de préciser leurs caractéristiques :



6. Le SGBDA doit mettre en application des modes de consommation (consumption modes)

Les modes de consommation déterminent l'ordre de traitement des événements; à savoir quels sont les événements considérés au sein de la composition d'événements. Ils existent plusieurs modes :

- *Recent policy* : la dernière occurrence d'un événement primitif - faisant partie de l'événement complexe - est traitée si l'événement complexe se produit.

- *Chronicle policy* : les événements sont traités chronologiquement.

- *Cumulative policy* : toutes les occurrences d'un événement primitif sont traitées si l'événement complexe se produit.

7. Le SGBDA doit gérer un historique des occurrences d'événements.

Le système doit supporter un contrôle d'événements ainsi qu'une sauvegarde de leur occurrence à travers un historique (event history) recensant les types d'événements (chaque événement primitif) et l'heure de leur occurrence (l'heure de la transaction).

8. Le SGBDA doit mettre en application un système de résolution de conflits.

Une politique de gestion des conflits doit être définie afin de gérer des règles qui se déclencheraient simultanément. L'utilisateur doit pouvoir spécifier différentes priorités en cas de conflits de règles pour déterminer un ordre logique de traitement et contrôler la concurrence de leur exécution. Cependant, si l'utilisateur ne veut pas définir une politique de résolution de conflits, le SGBDA doit être capable de déterminer certaines priorités dans l'exécution de règles.

9. Le SGBDA doit supporter un environnement de développement.

Il est évident que le SGBDA puisse disposer d'un langage de définition de règles, intégré dans le DDL (Data Definition Langage) de la base. En outre, afin d'assister tout développeur utilisant un SGBDA, des d'outils de développement semblent s'imposer ou du moins, sont intéressants à proposer :

- Un browser et analyseur de règles : un browser permet d'inspecter l'ensemble des règles existantes. Un outil permettant de déterminer si une règle a déjà été définie est essentiel. L'analyseur est un outil offrant la possibilité de vérifier certaines propriétés de règles. Notons encore que si le SGBDA supporte l'exécution de règles en cascade, il est essentiel de s'assurer que l'exécution se terminera quelles que soient les circonstances. De cette manière, le principe de terminaison assure que l'ensemble des règles est en accord avec le comportement réactif désiré.

- Un créateur de règles (designer) : Il s'agit d'un outil assistant l'utilisateur dans la création de nouvelles règles ECA. Par un mécanisme de graphes, de dessins.., des assistants peuvent rendre la création de règles beaucoup plus conviviale.

- Un debugger : Cet outil permet de contrôler l'exécution de règles et d'apporter une aide importante dans la vérification des règles (vérification des résultats des règles en rapport aux comportements réactifs désirés).

- Un outil de maintenance : un outil de gestion de l'évolution des règles est important. La suppression ou la modification de règles doit être assurée pour maintenir des règles en rapport aux exigences externes désirées. Ces exigences peuvent être en fréquente mutation. Cet outil inclut un debugger afin de tester les modifications effectuées.
- Un service de traçabilité : ce service apporte une facilité qui rassemble les occurrences d'événements et les exécutions de règles. De cette manière, l'administrateur de la base est capable de déterminer quelles actions le SGBDA a déclenchées automatiquement.

- Un outils de gestion des priorités.
- Et encore bien d'autres…

Tous ces outils peuvent être séparés de l'environnement de développement du SGBDA ou peuvent être une extension des outils existants d'un SGBD ou d'un outil CASE.

10. Le SGBDA doit être réglable.

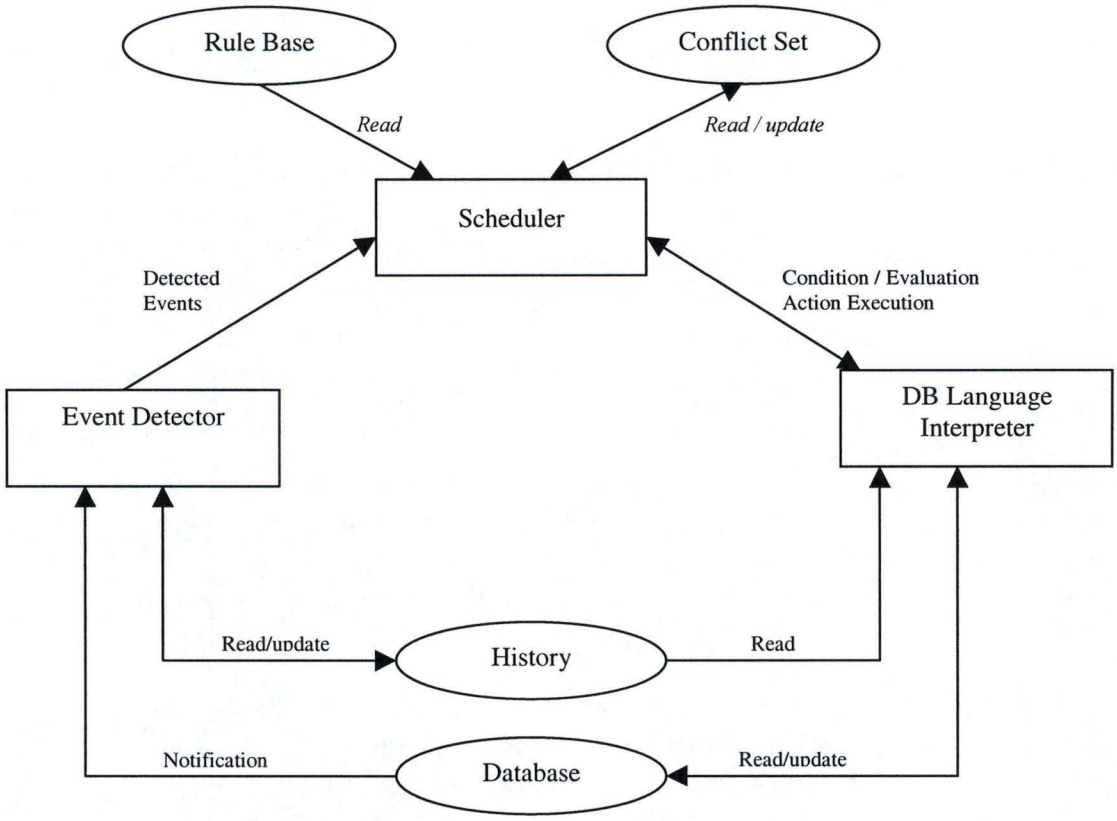
Un SGBDA doit être utilisable dans son domaine d'application. L'utilisation d'un SGBDA doit spécialement ne pas montrer des pertes de performance comparativement à d'autres solutions équivalentes. Il est dès lors important de fournir une bonne méthodologie de design et d'implémentation des règles pour atteindre des performances optimales.

1.3.2 Résumé des caractéristiques d'un SGBDA

- *Un SGBDA est un système de gestion de bases de données (SGBD).*
- *Un SGBDA applique le modèle ECA (Événement, Condition, Action).*
- *Un SGBDA doit supporter la gestion des règles ECA ainsi que leur évolution.*
- *Le SGBDA doit posséder un modèle d'exécution.*
- *Le SGBDA doit pouvoir proposer différents modes d'association (coupling modes).*
- *Le SGBDA doit mettre en application des modes de consommation (Consumption modes).*
- *Le SGBDA doit gérer un historique des occurrences d'événements.*
- *Le SGBDA doit mettre en application un système de résolution de conflits.*
- *Le SGBDA doit supporter un environnement de développement.*
- *Le SGBDA doit être réglable.*

1.4 La structure d'un SGBDA implémentant des règles actives

Nous allons présenter une structure relativement simple d'un SGBDA implémentant un système de règles en précisant comment celles-ci sont déclenchées en réponse à l'événements.



. ⬭ : représente une zone de sauvegarde d'informations
▭ : représente un processus du système

Structure d'un SGBDA implémentant des règles actives[2]

Analysons les différentes étapes de l'exécution d'une règle :

- Le SGBD informe le détecteur d'événements sur les occurrences dans la base de données pour lesquelles il doit apporter une attention particulière. Le détecteur peut, dès, lors lire et mettre à jour l'historique (History), qui sauvegarde une multitude d'informations sur les événements qui se sont produits.

- Le planificateur exige, de la part du détecteur d'événements, de l'information sur les événements détectés et examine dans la base de définition des règles, ce qui est associé à l'événement (action à accomplir). Les informations associées aux règles qui ont été déclenchées et pour lesquelles le planificateur a décidé de les exécuter plus tard, sont enregistrées dans le "conflict set".

[2] W. Paton, "Active Rules in Database systems", Monographs in Computer Sciences, springer, 1998, p.7.

- ❖ L'interpréteur est appelé par le détecteur chaque fois que la condition d'une règle doit être évaluée ou qu'une action doit être exécutée. Ce processus exige la lecture ou l'écriture d'informations dans la base de données. Il est également possible de couvrir, en examinant l'historique, certaines demandes d'information de la part de l'interpréteur sur les événements qui se sont produits. Ces accès ou mises à jour dans la base peuvent conduire à la détection d'autres événements qui forcent le processus à se reproduire. La boucle est bouclée.

Cette description est volontairement simplifiée puisque les systèmes de règles supportent des modèles d'exécution bien différents.

1.5 Classification des SGBDA

L'architecture d'un SBGDA détermine conjointement ces fonctionnalités et les éléments requis pour son implémentation. Certaines limites peuvent apparaître et poser certains problèmes sur l'architecture du SGBDA. En effet, les différentes caractéristiques que nous avons soulignées précédemment, permettent certains choix et une certaine liberté d'action dans leur intégration dans un SGBDA.

Pour rappel, un SGBDA est, par définition, une extension des SGDB passifs puisqu'il offre des fonctions actives en complément des caractéristiques des SGBD. Il existe plusieurs propriétés que les stratégies d'architecture doivent intégrer pour implémenter ces dimensions actives. Ces dernières peuvent avoir une influence sur les performances et les fonctionnalités du SGBDA.

Une tentative de classification[3] a été entreprise en distinguant le rôle que les SGBDA sont capables de remplir dans le système d'information (SI) ainsi que les fonctionnalités que le SGBDA doit proposer pour répondre aux exigences de certaines applications. De cette manière, la meilleur façon de procéder semble être de considérer toutes les classes de fonctions possibles et de déterminer leurs exigences. Sur base de cet ensemble, il suffit d'identifier les classes les plus appropriées pour le SGBDA analysé. Ces classes peuvent être considérées comme une superposition de couches que le systèmes doit posséder pour effectuer les traitements désirés.

Pour tenter de classifier les SGBDA à travers le rôle qu'il exerce au sein du système d'information, deux dimensions ont été retenues :

- ♦ Le rôle joué par le SGBDA dans le système d'information : "***Monitoring***" **ou** "***Control***".

"***Monitoring***" signifie que le SGBDA vérifie toutes les demandes d'information sur la base de données et éventuellement accomplit certaines actions élémentaires (interruption de transaction, la propagation de mise à jour..).
"***Control***" : Un SGBDA qui contrôle un système d'information est capable, outre le rôle de "Monitoring", de déclencher des fonctions externes telles qu'un programme d'application. De

[3] K. Dittrich, S. Gatziu et A. Geppert, "*The active database management system manisfesto : a rulebase of ADBMS features*", ACT-NET Consortium, 1997.

cette manière, il est compétent pour contrôler les actions de tout l'environnement de l'application (pas uniquement celui de la base de données).

- Le degré d'intégration du système d'information : ***Homogène*** *ou* ***Hétérogène***.

Le degré d'intégration entre le SGDB et les fonctionnalités actives constituent une dimension essentielle permettant de distinguer les architectures. En effet, en fonction du degré d'intégration, certaines fonctionnalités actives ne peuvent être implémentées ou du moins, être peu efficaces.

Un système est déclaré ***homogène*** lorsque tous ses composants sont des "couches" du SGBDA; c'est-à-dire qu'il partage un schéma commun et des bases de données communes. Dans les autres cas, le système est appelé ***hétérogène***.

En combinant ces deux variables, 4 classes de SGBDA se dégagent. Nous allons les décrire brièvement en dégageant leurs grandes spécificités en regard des caractéristiques des SGBDA décrites précédemment.

Notons encore que le système "Monitoring / hétérogène" ne fera pas l'objet de notre analyse car dans un système hétérogène (par exemple, en présence de plusieurs sources de données disparates), en lieu et place de surveiller des états valides des requêtes ou de la base, on aurait besoin de mécanismes actifs dans les systèmes externes plutôt que dans le SGBDA. En outre, ce type de SGBDA est incapable d'interdire, dans le système externe, certaines requêtes et au contraire, permettre l'exécution de certaines actions afin de rétablir la cohérence dans les sources de données externes. Ces actions serait alors classifiée comme un processus de contrôle. Nous n'aborderons, dès lors, pas ce type de système.

Dégageons les grandes spécificités des trois autres formes de SGBDA.

1.5.1 Un SGBDA "Monitoring" dans un SI Homogène

Dans un système homogène, le SGBDA a pour fonction de vérifier les états de la base de données. De cette manière, il peut discerner certaines requêtes des utilisateurs ou des applications, les vérifier par rapport à l'état de la base de donnée. Dans ce cas particulier, l'utilisateur ou l'application est complètement responsable de la définition des règles sémantiques limitant les actions sur l'entièreté du système d'information. Cependant, malgré que le SGBDA soit capable d'effectuer certaines actions (interrompre l'exécution de transaction, affichage de messages..), il ne possède aucune fonction de contrôle sur le système d'information.

Ce système est utilisé afin d'implémenter des taches classiques et relativement simples de gestion de base de données telles que les principes d'autorisation, de mise en œuvre de contraintes relativement simples de cohérence…

Si on examine l'intégration du modèle ECA au sein de ce système :

- Les types d'événements que le système peut détecter sont déterminés par le modèle des données; en particulier, il n'est pas nécessaire que ce soient des événements composés.

- Les conditions se situent au niveau des prédicats sur l'état de la base ou des requêtes.
- Les actions sont principalement des commandes DML.

Plusieurs modes d'association peuvent être employés. Ils dépendent principalement de leurs fonctionnalités. Pour les autorisations, le mode d'association employé est immédiat tandis que le mode différé est choisi pour la gestion de la cohérence.

1.5.2 Un SGBDA "Control" dans un SI Homogène

Toutes les propriétés du système, décrites précédemment sont intégrées dans ce type de système. Cependant, il offre une plus large ouverture puisqu'il est capable de contrôler, non seulement, la base de données mais également son environnement. Ce système est capable de contrôler les applications intégrées. Tous ces programmes d'applications utilisent le même schéma et le même modèle de transaction.
Dans un système d'information, le rôle d'un SGBDA de ce type est d'appliquer les règles stratégiques associées au domaine d'application étudié. Ceci implique que le SGBDA est capable d'implémenter sous la forme de règles ECA, l'information relative à l'environnement de l'application.

Il possède également la capacité de détecter tous les états ou les séquences d'événements du système d'information (SI) afin de réagir automatiquement. En effet, toutes les applications sont étroitement intégrées et les mécanismes "actifs" font partie intégrante du système homogène.

L'intégration du modèle ECA est la suivante :

- Les types d'événements sont plus importants puisque les événements composés, les restrictions d'événements et des intervalles de contrôles sont nécessaires. Les événements composés sont indispensables pour contrôler et vérifier les situations complexes dans l'environnement de la base de données.

- Les conditions peuvent être une fonction booléenne incluant des prédicats sur l'état de la base et des requêtes.

- Les actions sont des commandes DML mais elles peuvent inclure des appels à des programmes extérieurs

Un large éventail de règles sémantiques doit être intégré. Ceci implique que les modes d'association peuvent être immédiats, différés voire découplés.

1.5.3 Un SGBDA "Control" dans un SI Hétérogène

Ce dernier type de SGBDA est capable d'intégrer des systèmes hétérogènes et autonomes. Les mécanismes actifs peuvent permettre au SGBDA d'effectuer des contrôles sur la partie hétérogène mais plus difficilement sur le système intégré. Ce type de SGBDA (grâce à ces fonctionnalités) est appliqué dans certains domaines tels que les systèmes avancés de gestion de workflow et les systèmes industriels de contrôle en temps réel. Possédant les propriétés des deux systèmes précédemment décrits, ce SGBDA permet de détecter des situations de manière beaucoup plus globale puisqu'il est capable de constater des événements émanant d'autres systèmes d'informations ou d'autres machines. Les mécanismes d'exécution des règles doivent être beaucoup plus puissants puisqu'il est possible qu'une action déclenchée ne soit plus exécutée sous le contrôle local du gestionnaire des transactions.

L'intégration du modèle ECA se décrit de la même manière que le système précédent. Cependant, les règles peuvent implémenter des interventions dans des domaines plus larges. Si un SGBDA de ce type est prévu pour des applications en temps réel, les contraintes temporelles doivent être spécifiées. Il en est de même pour la définition des règles qui doit inclure des actions de contingences.

1.5.4 Tableau récapitulatif [4]

Résumons les caractéristiques essentiels des trois types de SGBDA que nous venons de décrire.

Type de système / *Caractéristiques*	**SGBDA de rôle "Monitoring" dans un SI homogène**	**SGBDA de rôle "Control" dans un SI homogène**	**SGBDA de rôle "Control" dans un SI hétérogène**
Événements	Opérations DML, événements pas nécessairement composés	Opérations DML, événements externes, composés	Opérations DML, événements externes, composés
Conditions	Prédicats sur l'état de la base ou des requêtes	Fonction booléenne incluant les prédicats sur l'état de la base ou des requêtes	Fonction booléenne incluant les prédicats sur l'état de la base ou des requêtes
Actions	Actions DML, notification d'un utilisateur	Actions DML, notification d'un utilisateur, programmes externes	Actions DML, notification d'un utilisateur, programmes externes, actions de contingence
Gestion des règles	Création, Suppression Active, Désactivée	Création, Suppression, Modification, Active, Désactivée	Création, Suppression, Modification, Active, Désactivée
Modes d'association	Au moins immédiats, voire différés	Au moins immédiats, voire différés ou découplés	immédiats, différés ou découplés
Modes de consommation	Chronologique	Au choix mais incluant le chronologique	Au choix mais incluant le chronologique
Exécution	Sous contrôle local	Sous contrôle local	Pas entièrement sous contrôle local

[4] K. Dittrich, S. Gatziu et A. Geppert, "The active database management system manisfesto : a rulebase of ADBMS features", ACT-NET Consortium, 1997, p.9-10.

1.6 Résumé

Nous venons de voir comment les mécanismes de réponse automatique à la survenance de certains événements apportent une extension appréciable aux systèmes classiques de gestion de base de données. Ces techniques d'intégration de règles ont été motivées par la multiplicité des applications pour lesquelles les principes des comportements réactifs étaient utiles voire indispensables.

La définition d'un SGBDA a été présentée à travers ses principales caractéristiques. Nous avons également présenté comment un SGBDA peut être implémenté en utilisant le modèle ECA (Événement - Condition – Action) et comment ces règles sont décrites, exécutées et maintenues au sein de la base de données.

Enfin, nous avons tenté de classifier les différents types de SGBDA à travers deux dimensions :

- Le rôle joué par le SGBDA dans le système d'information : "*Monitoring*" ou "*Control*"
- Le degré d'intégration du système d'information *: Homogène ou Hétérogène.*

Notre analyse se poursuit par la présentation des règles ECA et leurs applications dans les SGBDA. Nous présenterons les grands axes de recherche associés à ces règles actives et voir leurs objectifs et leur intégration au sein des systèmes actuels.

Chapitre 2:

Les règles actives et leurs applications

2.1 Introduction

2.2 Utilités des règles ECA

Chapitre 2:

Les règles actives et leurs applications

2.1 Introduction

Analysant les règles actives, les programmeurs et les développeurs ont rapidement mis en évidence les caractéristiques et les potentialités de l'intégration des règles ECA dans les SGBD. Si on retient certaines préoccupations des SGBD telles que la vérification des contraintes d'intégrité et le contrôle de sécurité, on peut voir que les SGBD doivent répondre à certaines tentatives d'interrogation de la base et à certaines opérations sur les données. Le mécanisme des règles actives élargit les fonctionnalités des SGBD en leur intégrant de nouvelles formes de comportement permettant, de manière automatique, de répondre à certaines situations bien identifiées; et cela, sans pour autant devoir modifier le code source du système. Cette opportunité a directement conduit les chercheurs à la conclusion que ce système actif est une forme idéale, un bon prototype pour expérimenter de nouvelles fonctionnalités que des SGBD classiques ne pouvaient supporter auparavant. En effet, les règles ECA peuvent être un formalisme adéquat pour imposer des comportements dans un SGBD. Elles permettent donc de restreindre la liberté des utilisateurs, quant à la manipulation des données, mais également de renforcer le contrôle et la sécurité du système.

Si au départ, les règles devaient être générées manuellement par le programmeur ou le développeur de la base, au fur et à mesure de leur étude, les recherches se sont concentrées sur le développement de langages[1] de haut niveau pour spécifier de nouvelles sémantiques ainsi que des outils pour simplifier leur réalisation et leur intégration dans le système. En complément, d'autres recherches se sont concentrées sur les méthodes de génération automatique de règles. Pour chacun de ces langages, une attention particulière doit être apportée pour déterminer comment identifier les événements pour lesquels des règles doivent être spécifiées, quelles conditions doivent être vérifiées suite à l'occurrence de l'événement et enfin, quelles actions exécutées en cas de vérification des conditions de déclenchement. Sans entrer dans les détails de ces langages, nous allons décrire quelles peuvent être les principales utilisations de ces règles ECA dans les SGBDA.

Différentes références appuient notre réflexion. Cependant, les articles de S. Embury et P. Gray, "*Database Internal Applications*", Monographs in Computer Sciences, springer, 1998 et

[1] Il existe de nombreux de projets de développement sur ces langages , nous pouvons en donner certaines références pour plus d'informations :

- *Ariel* : E. Hanson. "*The ariel project. In Active Database Systems - Triggers and Rules For Advanced Database Processing*", chapter 3, pages 63-86. Morgan Kaufman Publishers Inc., 1996.
- *Chimera* : Fraternali P., Paraboschi S., Chimera : A Language for designing Rule Applications, Monographs in Computer Sciences, springer, 1998.
- *Chimera* : http ://www.elet.polimi.it/users/dei/sections/compeng/stefano.ceri/methodology/methodology.htm : organization of the IDEA methodology.
- *Ode : N. Gehani, H.V. Jagadish, and O. Shumeli. Advanced Database Systems, Chapter 1 : COMPOSE : A System For Composite Specification And Detection". Springer, LNCS 759, 1993.*
- *Samos* : SAMOS, Gatziu S. et Klaus R. Dittrich, "*Samos*, Active Rules for Databases, Springer Verlag, New-York, , 1998, http ://www.ifi.unizh.ch/groups/dbtg/SAMOS/samos.html : The SAMOS project.
- Liste des projets actuels : http ://www.ida.his.se/ida/adc/bibl/complete.html.

de Kotz-Dittrich A. et Simon E., "*Active Database system, Expectations, Commercial Experience and beyond*", Monographs in Computer Sciences, springer, 1998, constituent la base de notre analyse.

2.2 Utilités des règles ECA

Trois grands axes ont été étudiés par rapport au modèle ECA et constituent actuellement une importante source de recherche. Cependant, les règles ECA peuvent couvrir une multitude d'autres domaines. En effet, les règles permettent d'apporter une forte personnalisation des comportements au sein des SGBDA en fonction de leur domaine d'application : la finance (bourse, secteur bancaire, société d'investissement), le secteur industriel (contrôle de processus de production, de gestion de production..), l'informatique (contrôle de réseau, gestion de l'information) ... Nous allons déterminer les principales fonctions que remplissent les règles actives directement en rapport avec le système dans lequel elles sont implémentées.

2.2.1 Les règles ECA pour la maintenance de l'intégrité

L'intégrité d'une base de données est un souci majeur dans une base de données. Toute opération mettant en péril cet objectif, doit être supprimée ou rectifiée dans les plus brefs délais.

Une base de données est déclarée valide lorsque toutes les contraintes d'intégrité sont respectées (contrainte de cohérence). Ce critère de cohérence a toujours été une priorité car il faut assurer que les spécifications déclaratives formalisant les conditions ne doivent, en aucun cas, être violées et cela, sans pour autant, détériorer les accès et les performances de la base de données. Si ces contraintes déclaratives peuvent être relativement simples (l'âge d'un client est compris entre 0 et 125 ans, le type d'action est "obligations, actions, sicav, .., call, swap"...), celles-ci peuvent être beaucoup plus complexes car elles peuvent combiner plusieurs attributs et plusieurs associations.

Les règles ECA ont été introduites comme un mécanisme adéquat pour implémenter les deux principaux aspects de la gestion de l'intégrité : la détection efficace des contraintes d'intégrité et l'exécution de certaines mises à jour qui permettent de restaurer l'intégrité de la base. Analysons plus en détails ces deux aspects :

a) Les règles ECA de vérification de contrainte

La vérification des contraintes est un souci permanent au sein d'une base de données. La vérification d'une contrainte peut être assimilée à une simple évaluation d'une requête qui spécifie les violations de la contrainte. L'intégrité est donc préservée si le résultat de la requête est vide. Cette forme de vérification de contraintes peut s'avérer peu efficace. Cependant, le principe de simplification[2] qui permet de spécialiser la vérification de contrainte, peut s'avérer plus performant. Explicitons ce principe pour une opération de mise à jour. Il se décompose en trois étapes:

[2] Nicolas J-M, "Logic for Improving Integrity Checking in Relational Databases, Acta Informatica, Vol 18 : 227-240, 1992.

Identification des opérations pouvant violer certaines contraintes

Une base de données valide peut devenir incohérente, par exemple, suite à une opération d'UPDATE. Dès lors, les contraintes doivent être vérifiées après toute mise à jour. Il en est de même pour les opérations d'insertion et de suppression.

Identification des mises à jour "non triviales"

Les opérations de mise à jour qui peuvent causer la violation d'une contrainte donnée, sont un sous-ensemble de toutes les mises à jour possibles sur la base. En effet, certaines mises à jour ne violent pas la contrainte analysée; elles sont appelées "triviales". C'est pourquoi, pour chaque update, seules les mises à jour non triviales doivent être contrôlées pour une mise à jour donnée.

Vérification des données mises à jour

Si la base de données est cohérence avant la mise à jour, il est seulement nécessaire de vérifier la cohérence des données affectées par la mise à jour et non toutes les données de la base.

Cette spécialisation permet de simplifier la vérification de contraintes en précisant pour chaque opération, les contraintes à vérifier. Ces caractéristiques permettent de penser que les règles ECA sont appropriées pour implémenter ce type de vérification. En effet, les règles ECA peuvent être générées sur base des événements déclencheurs associés aux opérations d'update non triviales et pour lesquelles les conditions représentent la vérification de la violation de la contrainte. L'action associée doit être spécifiée en fonction des circonstances ou de la volonté du programmeur (annulation de la transaction, correction de l'erreur ..). Des réserves doivent être apportées à cette vision un peu simpliste car la complexité du langage des contraintes, le nombre de règles à générer pour chaque contrainte, la difficulté de définir les événements déclencheurs et le nombre de simplifications forment autant d'obstacles qu'il est parfois difficile, voire impossible de surmonter sans nuire soit à la performance du système, soit à la clarté de modélisation.

b) Les règles ECA de la correction de contraintes

Ayant analysé les événements et les conditions des règles ECA, l'action à exécuter doit être également analysée. Si la principale fonction de la règle est d'éviter un état invalide de la base de données, l'action doit éviter, soit une mise à jour illégale (en cas de déclenchement de règles avant que la mise à jour ne soit effectuée (trigger de type before[3])), soit annuler (rollback) les effets de la transaction (si la règle est déclenchée après la mise à jour (trigger de type after)).
Cependant, une autre solution est envisageable. Elle permet au développeur de la base de spécifier, lui-même, l'action à exécuter. Cette exécution peut effectuer une mise à jour dont le but est de restaurer la cohérence de la base. Supposons la contrainte que toute opération boursière ne peut dépasser 10.000.000 francs, si une insertion tente d'ajouter une transaction d'un montant supérieur à la limite fixée, la règle peut forcer que le montant inséré soit égal à 10.000.000 francs et qu'un message soit transmis à l'utilisateur. De cette manière, la

[3] Cfr p.55

transaction n'est pas annulée et est valablement introduite dans la base. Le principe de cohérence est ainsi respecté. Cette technique a été appelée "*integrity repair*" ou "*transaction repair*".

Afin de pousser encore plus loin le raisonnement, une solution plus performante encore serait que le compilateur de contraintes puisse, lui-même, générer les actions de restauration à partir de la spécification des contraintes déclaratives. En effet, tel que le suggère Urban[4] , il est possible de générer des règles de réparations automatiques d'intégrité (Event – Condition-Repair Action). Cependant, le processus n'est pas complètement automatique parce qu'il peut exister plusieurs réparations possibles pour chaque contrainte analysée. Après la génération des règles, le créateur de la base de données peut choisir en fonction des solutions possibles, la ou les solutions qui lui conviennent le mieux. En complément, lors d'une erreur d'exécution, l'utilisateur peut, en fonction du développeur, intervenir afin de choisir parmi les solutions proposées, celle qui satisfait le mieux à ses exigences; tout en respectant les règles d'intégrité. L'inconvénient de cette approche est que les règles peuvent avoir des comportements "anormaux" car il est possible que la réparation d'une contrainte engendre d'autres violations de contraintes. Ce n'est pas forcément grave mais la difficulté peut se poser lorsque le processus de correction d'une de ces contraintes entraîne la violation de la première contrainte. Cette situation engendre un cycle infini de déclenchement de règles. Une solution de résolution peut être une approche hiérarchique où les règles sont associées à un niveau particulier du processus et elles ne peuvent être déclenchées en dehors de leur niveau.

Un autre problème peut apparaître lorsque la correction d'une contrainte est réalisée automatiquement par le système (sans intervention de l'utilisateur). En effet, tel que le souligne Gertz, "*The drawback of active rule approaches to repair is that they in general realize an autonomous repair of constraint violations. Though the user can choose between automatically derived repairing triggers at compile-time, these trigger are kept fixed at runtime. Once repair is triggerred in an inconsistent database, there is no way to interact with the repair process. Furthermore, often a repair of violations may introduce new violations which are then automatically repaired, and so on. Hence it is difficult for the user to identify why what happened. Interesting questions are also what happens if the result state does not reflect the user's intention or the application requirements? How can he choose between possible alternatives repairs.*" [5]

Enfin, soulignons encore le problème de la réparation de la violation associée à l'occurrence de plusieurs contraintes et qu'une seule stratégie de réparation doit être menée pour résoudre toutes les contraintes.

Malgré ces quelques problèmes, le modèle reste une solution relativement efficace pour assurer une vérification des contraintes et maintenir la cohérence de la base de données.

c) Les règles ECA pour la vérification de contraintes temporelles

Les contraintes temporelles expriment une séquence de changements d'états. Les contraintes suivantes sont de ce type : l'âge d'un client ne peut qu'augmenter; un client qui a eu deux

---

[4] S. Urban, A Karadimce et R. Nannapaneni, "Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database, In 8th Intl. Conference on Data Engineering, p 565-572, Phoenix, Arizona, 1992. IEEE Computer Sciences.

[5] Gertz, "Specifying Reactive Integrity Control for Active Database", in Widom and S. chakravarthy, editors, Proc. Of 4 th Intl Workshop on Reseach Issues in Data Engineering, Houston, Feb 94, p.42.

déficits de compte injustifiés, ne peut plus effectuer des transactions; seuls les clients ayant un montant d'épargne supérieur à 10.000.000 de francs, l'année dernière, peuvent demander un taux préférentiel.
En général, les contraintes de ce type sont déclarées via une logique temporelle qui exprime, avec des opérateurs, le fait que certaines logiques étaient vraies dans un état précédent, ou ont été vraies depuis qu'une autre logique est devenue vraie. Toman et Chomicki[6] ont donné une approche théorique afin de modéliser ces situations. Ils utilisent un ensemble de règles ECA pour maintenir un ensemble de valeurs qui satisfont certains prédicats temporels utilisés pour la vérification d'une contrainte temporelle. Si nous prenons notre deuxième exemple de contraintes temporelles, trois règles ECA seront générées pour vérifier la contrainte et deux ensembles de valeurs seront créés. La première règle prend en compte le fait qu'un client a un déficit de compte non justifié pour la première fois et enregistre ce fait dans la première valeur stockée. La deuxième règle se déclenche lorsqu'un client enregistré lors de la première étape effectue un deuxième déficit non justifié et enregistre une valeur dans le deuxième ensemble de valeurs. La troisième règle se déclenchera lorsqu'un des clients (dont les deux valeurs sont remplies) tentera d'effectuer une transaction. La conséquence est que la règle exécute une action d'annulation de la transaction associée et envoie un message à l'opérateur.

d) Les règles ECA pour la vérification de contraintes dans des bases de données distribuées

Sans entrer dans les détails, les contraintes devant être respectées dans les bases de données distribuées, sont principalement des contraintes de dépendance d'existence (la valeur d'une donnée doit exister dans une base si une donnée associée est présente dans une autre base) ou de dépendance de valeurs (la valeur d'une donnée dans une base est déterminée par la valeur d'une donnée associée dans une autre base). Ceri et Widom[7] proposent que des règles ECA soient générées, à partir de ces contraintes, de la même manière que les contraintes de bases non distribuées. Cependant, la condition doit comprendre une requête d'interrogation à distance afin de déterminer la valeur de la donnée distante et l'action de la règle doit propager les effets opérés vers toutes les bases concernées par une transaction à distance.

2.2.2 Les règles ECA pour la gestion des vues dans une base de données

Jusqu'à présent, nous avons, de manière implicite, travaillé avec ce qu'il est convenu d'appeler des "tables de base" permanentes. Une "table de base" est une table qui existe réellement en ce sens où lui correspondent des enregistrements stockés sur le support physique contenant la base. L'adjectif permanent indique que, dès qu'une table est créée, elle persiste dans la base de données jusqu'à ce qu'elle soit explicitement effacée. Cependant, dans les SGBD relationnels, la notion de schéma externe correspond au concept de tables dérivées qui peuvent être de deux types : les photographies et les vues.

Une photographie est une table contenant des données déduites à partir des tables de base (ou d'autres photographies. Elles sont stockées physiquement dans la base. Deux inconvénients sont constatés : d'une part, une perte d'espace est constatée puisque des données peuvent être redondantes et d'autres part, la difficulté de cohérence entre la photographie et les tables de la

---

[6] Toman and Chomicki, "Implementing Temporal Integrity Constraints Using An Active Database, in Widom and S. chakravarthy, editors, Proc. Of 4 th Intl Workshop on Reseach Issues in Data Engineering, Houston, Feb 94, P87-95.

[7] Ceri et Widom, "Managing Semantics Heterogeneity with production Rules and Persistent Queries, 19th Intl Conference on Very Large Databases, P227, Springer, 1992.

base est difficilement maintenue puisque toute modification intervenant dans les tables doit être répercutée sur les photographies.

Les vues sont des tables dérivées dynamiques, véritable fenêtre sur la base de données. Elles ne sont pas stockées physiquement dans la base. Dès qu'une vue est définie, on peut l'utiliser comme s'il s'agissait d'une table de base. Il s'agit, en quelques sortes, de tables virtuelles générées sur base des tables classiques. Elles permettent de présenter, en une seule représentation, les données de plusieurs tables en fonction de certaines relations spécifiées.
C'est ainsi que l'on peut exécuter des requêtes de recherche sur une vue, définir des droits d'accès, ou encore, définir d'autres vues à partir de celle-ci. Rien ne permet à un utilisateur de faire la distinction entre une table et une vue. Une vue peut être utilisée pour masquer la complexité du schéma de la base de données et ainsi fournir des "versions limitées" de schéma complet de la base en fonction des besoins des utilisateurs. Les vues permettent d'apporter une certaine sécurité et préserver une confidentialité des données puisque les utilisateurs, en fonction des privilèges, ont accès uniquement aux vues (partie du schéma général) pour lesquelles ils possèdent les droits. De cette manière, il y a indépendance de l'utilisateur vis-à-vis de l'organisation logique globale des données

Deux problèmes sont à considérer : comment assurer une recherche optimale des données de la vue et comment gérer les tentatives de mises à jour au travers des vues?

Pour résoudre le premier, il suffit de matérialiser le résultat de la vue en le sauvant physiquement. Cette technique élimine ainsi le problème de régénération de la vue à chaque accès. Cependant, elle reporte le problème sur la cohérence entre la vue et la base en cas de mise à jour. C'est dès lors un problème de maintenance de la vue qui doit être analysé. En outre, la vue doit être également mise à jour lorsque, dans la base de données, les informations qui lui sont associées, sont modifiées.

Les mises à jour des vues constituent une problématique relativement complexe car il n'est pas toujours possible d'effectuer des mises à jour au travers des vues. En effet, pour effectuer ces opérations, il doit être possible de propager la mise à jour sur les tables sources. Logiquement, la vue est telle qu'il existe une correspondance biunivoque entre les lignes de la vue et celles d'une seule table source, les modifications et les suppressions ne posent pas de problèmes et peuvent être propagées. Au-delà de cette règle, il est relativement évident qu'il existe des vues intrinsèquement non modifiables. Si la vue est créée par l'expression d'une fonction de calcul de somme ou de comptage, une mise à jour des résultats de ce calcul est bien évidemment impossible à répercuter. Les constructeurs des SGBD ont, à cet effet, émis un ensemble de conditions que doit respecter une vue pour être modifiable :

- La définition de la vue ne peut contenir les mots réservés : **JOIN, UNION, INTERSECT** ou **EXCEPT**. La vue est donc définie à partir d'une seule table source.
- La clause **SELECT** de la définition de la vue ne peut contenir de **DISTINCT.**
- La clause **SELECT** ne peut contenir que des références aux colonnes de la table source (pas de **SUM, COUNT**,...).
- La clause **FROM** contient exactement une table ou vue elle-même modifiable.
- La clause **WHERE** ne peut contenir une sous-question dont la clause **FROM** possède une référence à la même table que celle de la clause **FROM** citée au point précédent.
- La définition de la vue ne peut pas contenir de clause **GROUP BY** ou **HAVING.**

Une vue qui ne satisfait pas les conditions citées est appelée en **lecture seule (READ-ONLY)**. Cependant, les SGBD actuels possèdent toutes des variantes plus ou moins restrictives de ces différentes règles. Oracle permet, sous certaines conditions, qu'une vue de jointure puisse être modifiée.
Les règles ECA peuvent s'intégrer dans ces problématiques afin d'apporter certains propositions de solutions.

Une des premières applications des règles ECA implémentant le mécanisme des vues montre qu'il est possible d'intercepter les accès aux données d'une vue déclarée et de dévier les demandes de renseignements vers les données concrètes sur lesquelles la vue est définie. Ces règles peuvent être générées directement à la définition de la vue à partir de la spécification de la vue. Ceri et Widom[8] proposent une approche relativement simple pour la génération automatique de règles associées aux vues, dans laquelle les règles actives sont générées pour chacune des tables concrètes pour des événements d'insertion, de suppression et de mise à jour. Une solution plus efficace serait d'associer ces règles à des événements qui sont uniquement associés à des mises à jour dans la base qui est à l'origine des changements dans le contenu de la vue.

En complément, comme le souligne Stonebraker[9], les règles actives peuvent être utilisées pour résoudre le problème des mises à jour des vues en leur associant une politique particulière de mise à jour. En effet, une règle est créée pour chaque mise à jour possible d'attributs, de relations ou de classes virtuelles. L'action associée est d'exécuter la mise à jour qui doit être effectuée sur les données de la base en correspondance à la mise à jour de la vue. On voit directement que cette manière de procéder est lourde, nécessite des choix de la part du générateur de règles qui ne sont pas forcément les mêmes que ceux de l'utilisateur et qui ne peuvent pas s'appliquer à des vues trop complexes.

Les règles actives peuvent être également utilisées pour implémenter différentes stratégies de projection de vue. Le problème est de trouver une manière assurant que la vue réalisée reste cohérente lorsque les données de la base associée à la vue sont mises à jour. Les vues doivent donc être maintenues par rapport à toutes modifications des données de la base. Les règles actives peuvent être utilisées à cet effet. Imaginons une règle active générale de la forme :

SUR <Mise à jour de la base de données>
SI <Ensemble des changements à opérer dans la vue> = Change **ET** Change <> ∅
FAIRE <Mise à jour de la vue avec Change>

Les règles similaires peuvent être écrites pour des opérations de suppression.

La mise à jour des vues constitue une problématique très complexe et très difficile à mettre en œuvre car les conditions de mise à jour peuvent dépendre d'une multitude de facteurs qui peuvent s'opposer.

---

[8] Ceri S. et Widom J., "*Deriving production Rules for incremental View Maintenace*, 17th Intl Conference on Very Large Databases, P577-589, Morgan Kaufmann, 1991.
[9] Stonebraker M., Jhingran A., Goh J. et Potamianos, *"On Rules, Procedures, Caching and Views in DataBase Systems"*, in H. Garcia-Molina and H.V. Jagadish, editors, Proc of SIGMOD 90, P281-90, Atlantic City, May 1990.

2.2.3 Les règles ECA pour l'intégration de données dans des bases distribuées

Si le principe des vues est une manière de présenter une vision plus restreinte du schéma complet de la base, il est également utilisé dans le cadre de l'intégration des données dans des bases distribuées. De manière naturelle, les règles actives, soulignées au point précédent, peuvent être utilisées pour implémenter le mécanisme des vues dans ce cas particulier. Le rôle des règles est un peu différent car leur principal but est d'apporter un mécanisme assurant la gestion de la communication entre les différents composants autonomes (détection des besoins de communications, envoi de message..). Pour les lecteurs désirant de plus amples informations sur ce point, le projet WHIPS[10] constitue une approche intéressante de cette problématique.

2.2.4 Les règles ECA pour la gestion des transactions avancées

Le potentiel des règles ECA a été analysé afin de les intégrer dans le modèle des transactions avancées permettant de couvrir des fonctionnalités de long terme. Les modèles classiques de transaction, possédant les propriétés ACID (Atomicité, Cohérence, Isolation et Durabilité) , ne sont pas adaptés à ce type d'activité. En effet, lorsque des transactions doivent durer plusieurs jours, voire des semaines, il n'est pas possible d'utiliser les techniques classiques de verrouillage pour garantir le principe d'isolation de ces transactions. L'impact sur la disponibilité des données pour les autres transactions serait beaucoup trop contraignant. En outre, si la propriété d'isolation est un peu relâchée, la propriété d'atomicité est alors compromise car une commande d'annulation (*rollback*) ne peut plus forcément garantir un juste effacement des effets de la transaction. Pour effectuer ces activités très longue, la plupart des modèles permettent que les transactions puissent s'insérer au sein d'autres transactions et que des relations de dépendance soient spécifiées par rapport aux autres transactions indépendantes. De cette manière, si une transaction effectue un commit (ou un abort), les transactions liées doivent également effectuer la même opération. Des ensembles de transactions emboîtées peuvent se succéder pour planifier des activités complexes.

Des politiques de gestion de conflits doivent être créées en fonction des besoins de chaque activité. La notion de rollback est remplacée par un concept de compensation qui permet d'associer à chaque transaction, une transaction de compensation. L'exécution de cette transaction de compensation est la garantie de restaurer la base dans un état cohérent où les effets de la première transaction ont été annulés.

L'utilisation de règles pour spécifier et gérer des contrôles complexes découle de ces modèles de transactions. L'enchaînement des transactions peut s'effectuer par des règles où l'événement est la commande de confirmation de la première transaction. La condition peut être utilisée pour vérifier que les pré-conditions de l'ensemble des transactions sont satisfaites. L'action est simplement l'exécution de la transaction suivante. Conjointement, la dépendance des annulations entre les transactions peut être générée par une règle de la forme :

SUR <ABORT Transaction1 >
SI <Pré-conditions vérifiées>
DO <ABORT Transaction2 >

[10] Hammer J., Garcia-Molina H., widom J., Labio W. et Zhuge Y. , "The Stanford Data Warehousong Projet", Data Engineering Bulletin, 18(2) : p.41-48, 1995.

Cette technique reste hautement procédurale mais elle permet de fournir un mécanisme général et relativement flexible pour planifier des activités longues. Une meilleure approche serait de disposer de formalismes supérieurs (projet ACTA[11]) à partir desquels les règles pourraient être générées pour coordonner l'exécution de transactions.
Un type de longue activité qui peut être implémentée par des règles actives est le "workflow".

2.2.5 Autres utilisations des règles ECA

Outre ces trois pôles, les règles ECA peuvent être utilisées dans une multitude d'autres circonstances :

- Les règles ECA pour la gestion des erreurs : des règles de ce type peuvent se déclencher dans ces circonstances particulières. Les actions associées ont pour but d'exécuter un plan de contingence (rectification de l'erreur) afin que le processus se poursuive normalement.

- Les règles ECA pour un affichage dynamique des interfaces : les règles actives peuvent être utilisées afin, d'une part, de mettre à jour les affichages des interfaces des utilisateurs lorsqu'une mise à jour est effectuée par la base, elle-même, et, d'autre part, afin de copier adéquatement vers les autres interfaces, toutes mises à jour d'un utilisateur.

- Les règles ECA pour maintenir la cohérence lors d'une opération coopérative. Des règles actives peuvent être générées afin de prévenir des utilisateurs effectuant des opérations sur des données en cours de modification.

- Les règles ECA implémentant la production de règles en "forward chaining" : de cette manière, on peut voir comment la production de règles (règles condition-action) peut être transformée en un ensemble de règles ECA qui remplissent les mêmes objectifs.

- Et d'autres encore.

Les règles ECA peuvent aborder tous les domaines et toutes les fonctionnalités des SGBD (sécurité, performance, audit, statistique, contrôle...). Elles peuvent constituer un outil stratégique du système car leurs applications sont tellement diversifiées qu'elles peuvent être une technique de spécialisation des besoins des développeurs mais également des utilisateurs.

2.2.6 Les règles ECA et le constat de leur utilisation

Nous pouvons souligner que les règles actives permettent d'étendre les fonctionnalités des SGBD par la création de règles qui décrivent des comportements que l'on veut imposer et cela, sans aucune modification et compilation du code source. Un autre avantage appréciable est la facilité de gestion des règles (modification, création ou suppression). Cependant, développer une application en utilisant des règles actives n'est pas aussi évident qu'il n'y paraît. En effet, il est facile d'omettre de développer certaines règles en fonction de situations à considérer ou de développer des règles qui ne se déclencheront jamais. La création de règles associées à des vues et des données dans des bases distribuées, est rapidement complexe et est parfois

[11] Chrysanthis P. et Ramamritham K., "ACTA : The saga continues", Database Transaction Models for advanced Applications in Data Management Systems, Chap10, p349-397, Morgan Kaufmann Publishers, San Mateo 1992.

incompréhensible pour des utilisateurs non avertis. Pour pallier ce manque de clarté et de lisibilité, les recherches se sont dirigées vers la création de langages de haut niveau (de préférence déclaratif) où les règles déclaratives peuvent être générées automatiquement. Cette approche a l'avantage d'écarter l'utilisateur de la gestion, de la création et de la compréhension des règles. Notons encore que le mécanisme permettant d'étendre les comportements dans les bases de données, engendre le problème de créer un compilateur de ce langage qui est, en grande partie, indépendant du code du système existant.

Certains problèmes sont à mentionner : le contrôle des interactions entre les règles doit éviter des situations où l'exécution des règles engendre un état incohérent de la base de données (une transaction de réparation dont le but est d'annuler les effets des transactions précédentes, se déclenche après une série d'autres transactions de réparation) ou une non-terminaison de certaines transactions (une règle de mise à jour de vues qui cause une violation de contrainte, peut entraîner un cycle de déclenchement de règles). Pour éviter partiellement ce problème, des priorités de déclenchement doivent être imposées. Une analyse des conflits et de dépendances entre les règles existantes est assurément une dimension indispensable à intégrer dans le SGBD. Selon sa force d'action et d'analyse, la garantie d'exécution optimale des règles ECA peut s'établir.

La pratique des règles ECA a tendance à fournir certaines informations intéressantes. Les règles ECA semblent fonctionner correctement lorsqu'elles s'appliquent à des problèmes qui ne nécessitent pas des actions et des conditions trop complexes. Elles sont adaptées aux problèmes de maintenance des vues lorsque la base de données subit des mises à jour. En revanche, elles le semblent moins dans cas de réparation d'intégrité trop complexe. Un domaine où les règles ECA ont fait leur preuve et semblent être particulièrement performantes, est la gestion de la communication entre des composants distribués ou autonomes parce qu'en général, les règles, dans ces systèmes, sont relativement simples malgré que leurs actions puissent être complexes. Les règles ECA sont assurément un concept porteur pour l'avenir.

Après cette analyse théorique, nous allons aborder la mise en pratique de ces concepts à travers l'analyse de leur implémentation dans le langage PL/SQL d'Oracle.

Chapitre 3:

Le langage PL/SQL

3.1 Introduction

3.2 Caractéristiques du langage PL/SQL

3.2.1 PL/SQL : un langage de développement

3.2.2 PL/SQL : une facilité de développement d'applications

3.2.3 Oracle PL/SQL Release 8.0: nouvelles caractéristiques

3.3 Description du Langage

3.3.1 Les extensions des types de données

3.3.2 Structure et syntaxe des expressions classiques

Chapitre 3 :

Le langage PL/SQL

3.1 Introduction

Les triggers et les procédures stockées sont des mécanismes que l'on retrouve dans la plupart des SGBDA actuels. Ils sont relativement neufs et peu étudiés sur le plan théorique. Cette partie a pour objectif de nous éclairer sur leur pratique, leur utilité et leur implémentation dans les SGBD les plus connus. Plusieurs constructeurs proposent déjà leur utilisation. C'est le cas d'Oracle avec PL/SQL, de Sybase ainsi que SQL Server avec Transact/SQL et le SQL de RDB (depuis la version 6.0).

Le langage que nous avons choisi est le langage PL/SQL d'Oracle pour différentes raisons : Oracle est un système de gestion de base de données le plus courant et le plus utilisé dans le monde. Son langage PL/SQL est le résultat d'importantes recherches et a tendance à devenir un des standards de développement en matière de bases de données (actives en particulier). Enfin, des raisons professionnelles m'incitent à étudier ce langage. Dans ce chapitre, nous allons brièvement décrire ce langage et ces fonctionnalités afin de définir, par la suite, les procédures stockées et les triggers.

3.2 Caractéristiques du langage PL/SQL

PL/SQL est un langage de programmation procédural destiné au développement d'applications associées aux bases de données. Ce langage est une extension du langage SQL. PL/SQL peut tourner sur un grand nombre d'environnements différents et est actuellement incorporé dans le RDBMS[1] d'Oracle. Ce langage est structuré en blocs offrant ainsi, aux développeurs d'application, la possibilité de combiner la logique procédurale avec le langage SQL. Cette opportunité permet de satisfaire de nombreuses demandes complexes sur les bases de données qui ne pouvaient être exécutées auparavant d'une manière adéquate.

3.2.1 PL/SQL : un langage de développement

PL/SQL est un langage de développement qui fournit un accès à SQL, une portabilité, un mécanisme de sécurité et une facilité d'internationalisation.

- **Accès à SQL**

  Ce langage est la manière la plus aisée et la plus rapide d'accéder à SQL dans une base de données Oracle. Il supporte tous les types de données et les commandes SQL (DLL et DML). Cela réduit la nécessité de convertir des données échangées entre les applications et la base de données. Une intégration avec les outils d'Oracle et les interfaces de

[1] Relational DataBase Management System

programmation sont proposées. PL/SQL est une partie intégrante des outils d'Oracle tels que le "Develops/2000 et Designer/2000" et est intégré avec les interfaces de programmation telles que "Oracle Precompiler".

- **Portabilité**

PL/SQL permet aux développeurs de séparer leurs choix de plates-formes des décisions du développement. PL/SQL est supporté par Oracle sur toutes les plates-formes sur lesquelles Oracle offre son RDBMS et ses outils. C'est donc une portabilité au sein de l'environnement d'Oracle.

- **Sécurité**

PL/SQL offre un modèle de sécurité robuste en permettant aux administrateurs de fournir aux utilisateurs un accès à des données critiques sans leur permettre de modifier ces données par un accès direct aux tables elles-mêmes.

- **Facilité d'internationalisation**

PL/SQL facilite l'intégration et le développement d'applications devant être utilisées dans différents pays.

3.2.2 PL/SQL : une facilité de développement d'applications

PL/SQL améliore la productivité des développeurs et apporte des perfectionnements appréciables en matière de performances.

- **La productivité des développeurs**

Facilité d'utilisation : en fournissant des possibilités procédurales aux programmeurs SQL, PL/SQL réduit la nécessité de programmer des applications dans d'autres langages. De cette manière, dans les cas extrêmes, les développeurs ne doivent plus connaître qu'un seul langage.

Réutilisation de code : les procédures stockées PS/SQL permettent la réutilisation de codes en permettant, à plusieurs applications de partager une même procédure ou fonction.

Exécution sécurisée : un programme PL/SQL peut s'exécuter en toute sécurité. En effet, une vérification complète des types est effectuée lors de la compilation. Une autre caractéristique importante est la possibilité d'encapsulation.

- **Amélioration de la performance**

Les procédures stockées PL/SQL permettent que les applications soient exécutées sur le serveur. Cette opportunité permet d'éviter un trafic inutile d'informations entre le client et le serveur et d'améliorer les temps de réponse. Les procédures stockées qui sont compilées une et une seule fois, sont sauvegardées dans la forme exécutable. Leur

appel est donc exécuté d'une manière beaucoup plus rapide et efficace. Ces procédures peuvent être partagées avec les autres utilisateurs ; ce qui permet des gains de mémoire.

3.2.3 Oracle PL/SQL Release 8.0 : nouvelles caractéristiques

PL/SQL offre bien d'autres caractéristiques intéressantes dans sa nouvelle version 8.0. Elles se situent au niveau du découpage d'applications, du développement d'applications distribuées et du support dans les environnements multi-utilisateurs. La nouvelle version Oracle renforce l'extension des fonctionnalités du langage. Ces nouvelles caractéristiques sont principalement: PL/SQL supporte les objets d'Oracle8 comme des variables liées et ainsi il peut les utiliser pour implémenter des méthodes. Oracle 8 définit deux nouveaux types de collections (varray et nested Tables) qui permettent de fournir une représentation naturelle des relations [N-N] et [1-N]. Ces variables peuvent être utilisées dans les procédures PL/SQL. Les procédures externes PL/SQL fournissent une manière sûre et facile d'exécuter, sur le serveur, du code écrit dans un langage de troisième génération. Enfin, les performances d'appel ont été améliorées : un appel d'une fonction PL/SQL de la part de SQL est 40% plus rapide. Nous ne détaillerons pas ces fonctionnalités car elles ne font pas l'objet de notre analyse. Pour des informations plus complètes, nous proposons deux références qui abordent les caractéristiques complètes du langage sous la release 8.0 :

- Oracle PL/SQL Release 8.0, Feature Fac Sheet, http://www.oracle.com/database/documents/o8/pl/sql_newfeatures_o8_ov.html
- Oracle PL/SQL New Features with Oracle8 and Future Directions, An Technical white paper, http://www.oracle.com/database/documents/o8/pl/sql_newfeatures_o8_tpw.html

3.3 Description du Langage

Oracle permet d'accéder et de manipuler de l'information en utilisant des objets procéduraux appelés unités de programmation PL/SQL. Les procédures, les fonctions et les packages forment les éléments des unités de programmation PL/SQL. Le "PL/SQL engine" est l'outil principal qui permet de définir, de compiler et d'exécuter les parties de programme PL/SQL.

Le langage PL/SQL est un langage typé au sens large car il permet des conversions implicites. En effet, si un nombre N peut être passé comme argument à une procédure P conçue pour accepter uniquement une chaîne de caractères, Oracle opère implicitement la conversion. Les vérifications de typage se font lors de la compilation ou à l'exécution. Par exemple, lors de la compilation, une erreur est générée si l'on tente d'assigner un type composé à un type simple. Une erreur à l'exécution se produit si une variable de type NUMBER[2] et de valeur 256 est assignée à une chaîne de caractère de type VARCHAR2(2).

3.3.1 Les extensions des types de données

Dans un souci de performance, les types de données qu'il est possible de définir et d'utiliser en PL/SQL, sont très proches des types de données SQL et permettent une meilleure intégration des données entre la "mémoire centrale" et le "disque".

[2] Une variable de type NUMBER est implicitement convertie en une chaîne de caractères de longueur 2 (VARCHAR2(2)).

Il est tout à fait évident que les types prédéfinis en PL/SQL sont les mêmes que ceux qui permettent de déclarer les colonnes des tables SQL : *CHAR, NUMBER, VARCHAR, DATE, BINARY_INTEGER...* Tous ces types incluent également la valeur NULL.

Ce langage offre des extensions dans la définition des types en permettant diverses spécifications de référence.

- Une variable PL/SQL peut être déclarée du même type qu'une colonne d'une table SQL. Imaginons qu'une nouvelle variable NOM qui doit être déclarée du même type que le champs CLI_NOM de la table clients, la déclaration de cette variable est la suivante :
  NOM CLIENTS.CLI_NOM%TYPE;
  De manière générale, la syntaxe de déclaration d'une variable par référence à une autre variable est : *nom_variable nomtablesql.nomcolonne%**TYPE;***. Il est évident que l'élément référencé doit exister sous peine de générer une erreur de compilation.

- Partant du même principe, il possible de définir une variable PL/SQL qui doit être du même type qu'un enregistrement d'une table SQL. Cette déclaration permet de déclarer un enregistrement dont la structure déclarée est équivalente aux éléments de la table de référence. Soit INFOCLIENT, une variable devant être du même type qu'un enregistrement de la table *CLIENTS*. Elle est déclarée comme suit : INFOCLIENT CLIENTS**%ROWTYPE;**
  De cette manière, INFOCLIENT est un record composé des mêmes champs qu'une ligne de la table clients :
  INFOCLIENT.CLI_NUMID, INFOCLIENT.CLI_NOM, INFOCLIENT_CLI_PRENOM...
  La déclaration générale de ce type est : *nom_variable nomtablesql**%ROWTYPE;***

- Si l'on veut déclarer un record personnalisé au sens classique, il existe le type PL/SQL Record. Imaginons la déclaration d'un type de record *selectinfo* concaténant un nom, un enregistrement de la table *comptes* et une date. Nous pouvons utiliser ce type de record pour définir le type de la variable *informationclient*.

  TYPE SELECTINFO_CLIENT **IS RECORD**
  (NOM CLIENTS.CLI_NOM; COMPTE COMPTE%ROWTYPE; DATECOURANTE DATE);
  INFORMATIONCLIENT SELECTINFO_CLIENT;

- Enfin, le type PL/SQL TABLE permet de définir une variable qui possède une structure très proche de la table SQL associée. Ce type est l'équivalent des tableaux (array en Pascal). Par exemple, on peut définir le type CLIENTS_TYPE :

  TYPE CLIENTS_TYPE **IS**
  TABLE OF CLIENTS%ROWTYPE **INDEX BY** BINARY_INTEGER
  TESTTABCLIENT CLIENTS_TYPE.

  Les contraintes déclaratives qui s'appliquaient sur la table SQL ne s'appliquent plus sur la table PL/SQL. Cependant, il est permis de déclarer la contrainte NOT NULL sur des éléments d'une table PL/SQL, ainsi que sur des champs d'une structure de type PL/SQL RECORD.

3.3.2 Structure et syntaxe des expressions classiques

PL/SQL intègre les structures de contrôle (boucle, alternative,..) que l'on retrouve dans la plupart des langages impératifs :

IF *expression_booléenne* **THEN** ... **ELSE** **END IF;**	**LOOP** .... **EXIT WHEN** *condition* **ENDLOOP;**
FOR *index IN borne_inf..borne_sup* **LOOP** ... **END LOOP;**	***WHILE** expression_booléenne **LOOP*** ... ***END LOOP;***

Une documentation abondante explique, de manière complète, la syntaxe de tous les composants du langage. Elle est directement consultable à l'adresse http://www.oracle.com) En outre, un autre site Internet fournit la syntaxe BNF[3] du langage PL/SQL (version 2.1 for Oracle 7): http://cui.unige.ch/db-research/Enseignement/analyseinfo/PLSQL21/BNFindex.html

Remarque :
Les paramètres des procédures peuvent être de 3 types :

- **IN** : le paramètre de ce type est *read-only* : la procédure peut lire sa valeur mais ne peut pas la modifier. C'est l'équivalent d'un passage par valeur.
- **OUT** : le paramètre de ce type est *write-only* : la procédure peut y écrire un résultat mais ne peut pas y lire une valeur initiale. C'est l'équivalent d'un passage par résultat.
- **IN OUT** : La procédure peut lire le contenu du paramètre et peut également le modifier. C'est l'équivalent d'un passage par adresse.

Nous découvrirons encore d'autres caractéristiques du langage PL/SQL au cours notre travail. Nous les expliciterons directement en fonction de leur utilisation.

Ayant déterminé les grandes caractéristiques du langage PL/SQL, nous allons continuer notre analyse à travers l'étude des procédures stockées (stored procedures). Leur définition, leur description et leur implémentation en PL/SQL formeront les principaux points de notre réflexion.

[3] Backus-Naur Form notation.

Chapitre 4:

Les Procédures Stockées

4.1 Définition des procédures stockées

4.2 Types des procédures stockées

4.2.1 Les procédures

4.2.2 Les fonctions

4.2.3 Les packages

4.2.4 Les packages prédéfinis

4.3 Avantages des procédures stockées

4.3.1 Sécurité

4.3.2 Performance et productivité

4.4 Exécution des procédures stockées

Chapitre 4 :

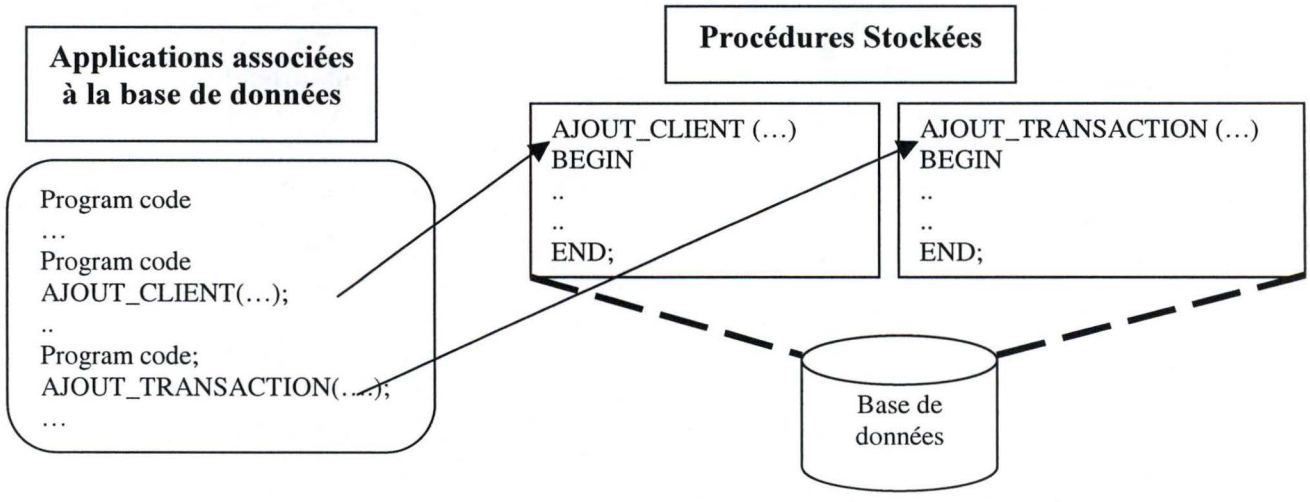
Les Procédures Stockées

4.1 Définition des procédures stockées

Enregistrées directement dans une base de données, les procédures stockées (stored procédure) sont des compositions d'une ou plusieurs instructions SQL compilées. Elles permettent de traiter certains problèmes de manière impérative et sont déclenchées par des événements ou par des applications. Prévues pour effectuer un traitement sur des données, les procédures stockées sont une extension du langage SQL standard.. En effet, il est possible de déclarer des variables, de leur affecter des valeurs et également d'utiliser des instructions de contrôle (IF, CASE, LOOP, FOR). Le langage PL/SQL qui permet de définir les procédures stockées dans Oracle, offre la possibilité d'appels récursifs (appel d'une procédure au sein d'une procédure).

En résumé, les procédures stockées sont des segments de codes procéduraux enregistrés dans la base de données. Au même titre que les tables SQL, elles sont, dès lors, considérées comme des objets de la base. Les procédures et les fonctions qui forment les procédures stockées sont des objets qui sont regroupés logiquement comme un ensemble d'instructions du langage SQL, voire PL/SQL, qui permettent en commun de répondre à certains traitements bien particuliers. Ces procédures et ces fonctions sont donc créées par un utilisateur et stockées directement dans la base de données pour une utilisation permanente. En effet, il est possible d'exécuter ces procédures ou fonctions de manière interactive grâce à certains outils spécifiques (SQL*Plus), ou encore, de les appeler explicitement, soit dans le code d'une application, soit dans le code d'une autre procédure ou d'un trigger.

Le schéma ci-dessous illustre deux simples procédures d'insertion stockées directement dans la base de données et qui font l'objet d'appel venant d'applications externes à la base. Le principe des procédures s'applique tout autant aux fonctions. Le principe des procédures ou des fonctions est identique à l'exception près qu'une fonction renvoie toujours une valeur unique à l'objet appelant alors qu'une procédure ne le fait pas.



Procédures stockées appelées par des applications externes

La procédure AJOUT_CLIENT que nous avons définie ci-dessus peut s'écrire :

```
PROCEDURE AJOUT_CLIENT( nom VARCHAR2, prenom VARCHAR2, numbroker NUMBER; date
DATE, compte NUMBER...) IS
BEGIN
INSERT INTO CLIENTS VALUES
        (CLIENT_NUMID.NEXTVAL[1], nom, prenom, numbroker, date, compte,...);
END;
```

Toutes les applications de la base de données peuvent faire appel à cette procédure ainsi définie. Notons encore qu'il est possible pour les utilisateurs privilégiés d'utiliser certains outils tels que les services du "Server Manager" ou "Enterprise Manager" d'Oracle pour exécuter directement la procédure d'ajout de client via une instruction :

```
EXECUTE AJOUT_CLIENT('dupont', 'marc', 0000010; SYSDATE, 250 0170204 58,...);
```

Le client Marc Dupond ainsi que toutes les informations relatives, sont de cette manière, encodés.

4.2 Types des procédures stockées

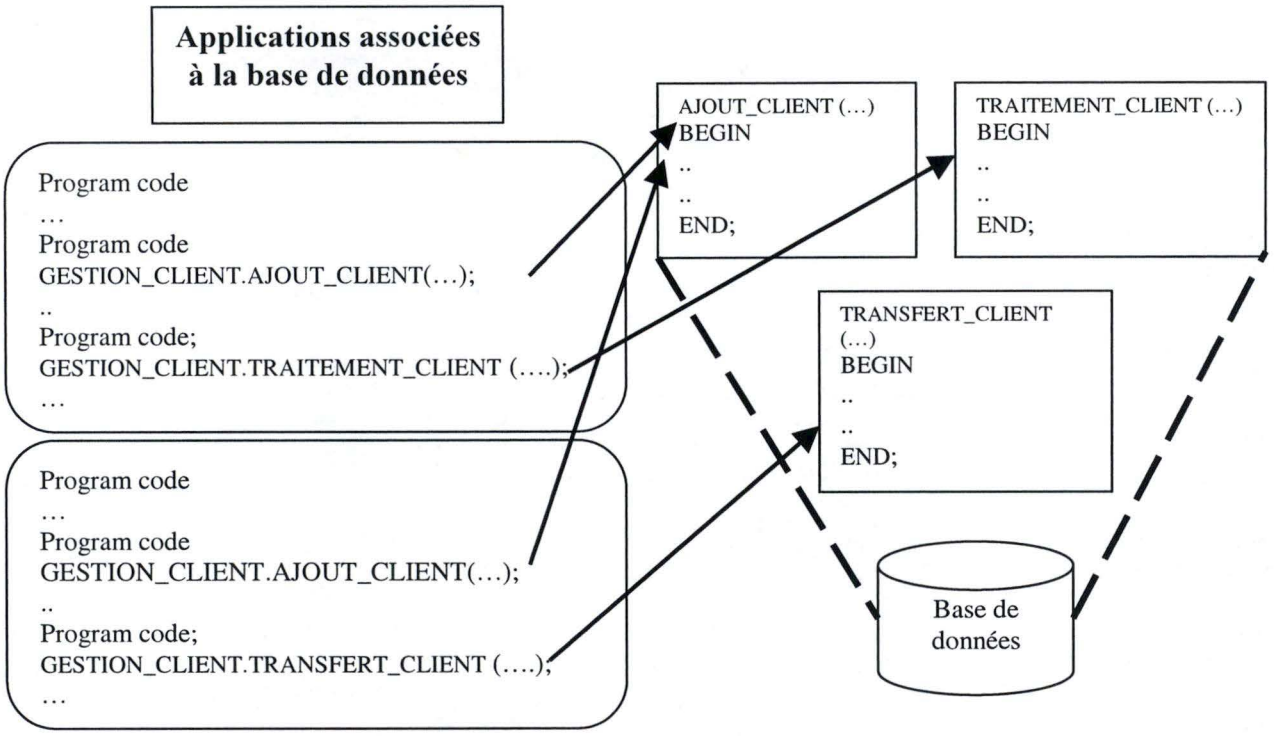
Oracle définit trois types de procédures stockées : les procédures (PROCEDURE), les fonctions (FUNCTION) et les "PACKAGES". Les procédures et les fonctions sont des portions de code autonome. Les packages encapsulent des procédures, des fonctions, des définitions de type et des déclarations de données . Ils ne peuvent dès lors jamais être appelés directement ; ce sont les fonctions ou les procédures insérées qui le sont.

Les procédures ou fonctions sont, de cette manière, désignées sous le nom de « sous-programme ». Chaque package est décomposé en deux parties : la première, dénommée PACKAGE SPECIFICATION permet de déclarer l'en-tête des procédures et fonctions du package associé ; la seconde appelée PACKAGE BODY contient l'ensemble des instructions associées. Les procédures, les fonctions et les packages sont des segments de programmes tout à fait distincts, qui sont compilés dans la base de données comme des objets testables indépendamment. Cette représentation se retrouve dans la plupart des langages impératifs[2].

Le schéma ci-dessous présente un package CLI_GESTION regroupant différentes procédures associées à la gestion des clients. Il regroupe, entre autres, les procédures AJOUT_CLIENT (ajout d'un client dans la base de données), TRAITEMENT_CLIENT (traitement d'un portefeuille d'un client particulier), TRANSFERT_CLIENT (transfert d'un client dans une autre filiale).

[1] Instruction permettant l'incrémentation automatique de la variable.

[2] Par exemple, les fichiers .h et .c dans le langage C.



Procédures stockées du package CLI_GESTION appelées par des applications.

Nous voyons que le package définit les diverses procédures. Les applications peuvent faire appel à ces procédures en spécifiant le nom de la procédure appelée, précédé du nom du package dans lequel elle se trouve. Nous verrons au cours de notre analyse que la technique de regroupement des procédures et des fonctions au sein de packages offre certains avantages de développement et de performance par rapport aux procédures stockées isolées.

Les procédures, ainsi que les fonctions, sont des objets de la base de données composés d'une série d'instructions SQL, voire PL/SQL. Elles peuvent être appelées par une application qui lui fournit les paramètres d'entrée (INPUT) et reçoit les résultats de l'exécution (OUTPUT). Les arguments des procédures ou fonctions sont facultatifs. Cependant, une fonction renverra toujours une valeur en résultat de son exécution.

Les procédures stockées peuvent faire appel à des procédures ou fonctions externes ; par exemple, des procédures écrites en C et enregistrées dans des librairies partagées.

Examinons maintenant plus en détails les trois types de procédures stockées. Notre analyse se concentrera sur les procédures stockées telles qu'elles sont définies par le langage PL/SQL.

4.2.1 Les procédures

Une procédure PL/SQL est un programme autonome qui est compilé au sein de la base de données. Elle peut accepter des arguments. Le schéma ci-dessous nous en décrit la structure :

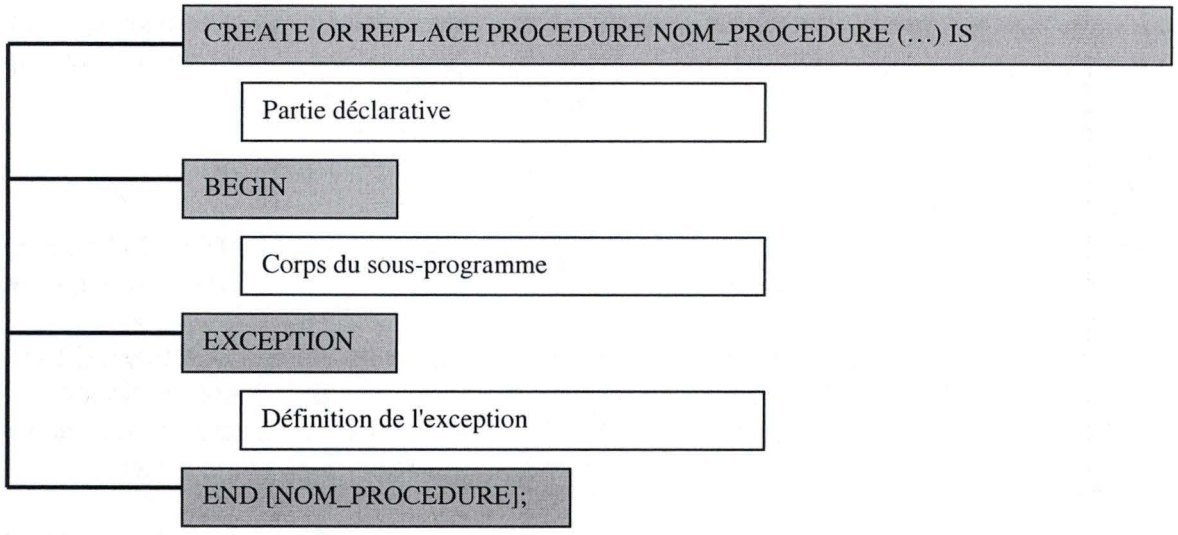


Schéma d'une procédure

Lorsque la procédure a été compilée, l'identifiant de la procédure correspond à la partie de l'instruction CREATE PROCEDURE. Le nom de la procédure devient le nom de référence à l'objet défini. Le type de l'objet est de manière évidente "PROCEDURE".

Trois parties forment le contenu d'une procédure.

- La partie déclarative :

Elle permet de définir des types, des variables et des constantes.

- Corps du sous-programme :

Cette partie contient l'algorithme logique implémenté selon la construction du langage PL/SQL. Ceci permet simplement de définir le traitement de la procédure en fonction des paramètres définis (s'il y en a). Cette partie peut faire appel à d'autres procédures ou fonctions.

- Définition des exceptions

Cette partie permet de gérer les erreurs pouvant survenir au cours de l'exécution de la procédure.

Exemple de procédure : soit une procédure DEBIT_CREDIT_COMPTE qui permet, sur base d'un numéro de compte d'un client, d'actualiser son compte en fonction du montant de débit ou crédit précisé. Un crédit est représenté par un montant positif, une débit par un montant négatif. Notons que pour une opération de crédit, si le numéro du compte (qui est identifiant de la table) n'existe pas, un nouveau compte est créé et qu'après une opération de débit, le montant du compte ne peut, en aucun cas, être inférieur à 1000 francs.

```
CREATE OR REPLACE[3] PROCEDURE DEBIT_CREDIT_COMPTE (numcompte NUMBER, montant NUMBER) IS[4]
/* la procédure accepte deux arguments : le numéro de compte et le montant de l'opération. */
/* Déclaration des variables /*
        old.solde  NUMBER;
        new.solde NUMBER;
BEGIN
/* corps de la procédure /*
        SELECT solde INTO old.solde FROM comptes
          WHERE compte_num=numcompte;
          FOR UPDATE OF solde;
          /* si aucun compte n'est trouvé, la partie exception continue le traitement via l'exception prédéfinie
          NO_DATA_FOUND qui va permettre d'insérer un nouveau compte si le montant est positif */
          new.solde := old.solde + montant;
          IF (new.solde >=1000) THEN
                  /* mise à jour du compte sur base de l'opération /*
                  UPDATE  comptes SET solde = new.solde WHERE compte_num=numcompte;
                  COMMIT;
          ELSE
                  /* annulation de l'opération et affichage d'un message d'avertissement  /*
                  DBMS_OUTPUT.PUT_LINE[5] ('Opération annulée, votre compte ne peut être inférieur à 1000
                  francs);
                  ROLLBACK;
          ENDIF
EXCEPTION
          WHEN NO_DATA_FOUND and (montant >0) THEN
          /* insertion du nouveau compte et de son solde correspondant au montant spécifié /*
                  INSERT INTO comptes (compte_num, solde) VALUES (numcompte, montant);
          WHEN OTHERS THEN  ROLLBACK;
/* fin de la procédure /*
 END DEBIT_CREDIT_COMPTE;
```

Nous pouvons remarquer que cette procédure combine des instructions SQL et PL/SQL.

Tout comme un programme classique, la démarche d'écriture d'une procédure stockée est la suivante :

1. Générer le code de la procédure directement via un éditeur ou utiliser les outils de développement existants (Outils CASE);
2. Compiler le code qui crée l'objet au sein de la base de données;
3. Tester la procédure selon tous les cas dans lesquels la procédure peut être utilisée. Ne pas négliger les cas extrêmes;
4. Corriger les instructions erronées et exécuter les étapes 2 et 3 jusqu'à la satisfaction des besoins.

Remarques :

Il est généralement conseillé de définir une procédure par tâche précise à accomplir. Créer une procédure permettant de réaliser plusieurs tâches simultanément conduit, dans la plupart des cas, à de longues procédures peu lisibles et peu performantes.
Sauf dans certains cas bien précis, il n'est pas nécessaire de réécrire des fonctionnalités fournies par le SBGD. Par exemple, il est inutile de définir des procédures pour mettre en

[3] L'option REPLACE permet, lors de la compilation, de remplacer l'objet dans la base de donnée si celui existe déjà.
[4] Il est également possible de remplacer IS par AS.
[5] Package prédéfini qui contient des procédures et des fonctions permettant de manipuler un buffer d'affichage.

vigueur certaines règles d'intégrité des données qui peuvent être appliquées par l'usage des contraintes déclaratives. Nous reviendrons sur ce point lors de l'utilisation des triggers dans le cadre des contraintes d'intégrité.

4.2.2 Les fonctions

Les fonctions sont équivalentes aux procédures si ce n'est qu'une fonction renvoie une valeur. A défaut, une erreur de compilation est générée.

Sur base de son identifiant (nom_fonction), un objet de type FUNCTION est créé à la compilation. Une fonction se compose, comme nous le montre le schéma ci-dessous :

- de la définition de la fonction, composée du nom de la fonction, de ses arguments éventuels et du type de la valeur de retour. Si la fonction ne possède pas d'arguments, les parenthèses ne sont pas nécessaires;
- de la partie déclarative de la fonction qui définit les variables, les types de données et les constantes;
- du corps de la fonction : c'est-à-dire du bloc d'instructions SQL et PL/SQL formalisant la logique de l'algorithme associé à la fonction. Il est possible, au sein de cette partie, de faire appel à d'autres procédures ou fonctions;
- des exceptions assurant une personnalisation des erreurs pouvant apparaître au cours du traitement. Cette partie est facultative.

L'algorithme de la fonction doit obligatoirement s'assurer que tous les alternatives possibles conduisent à une instruction de retour de valeur (RETURN), même dans le cas d'une exception. Sans cette valeur de retour, une erreur à l'exécution se produit. En outre, une fonction ne peut retourner une valeur ou une expression qui n'est pas du type de la valeur de retour de la fonction (ou qui ne peut être implicitement convertie).

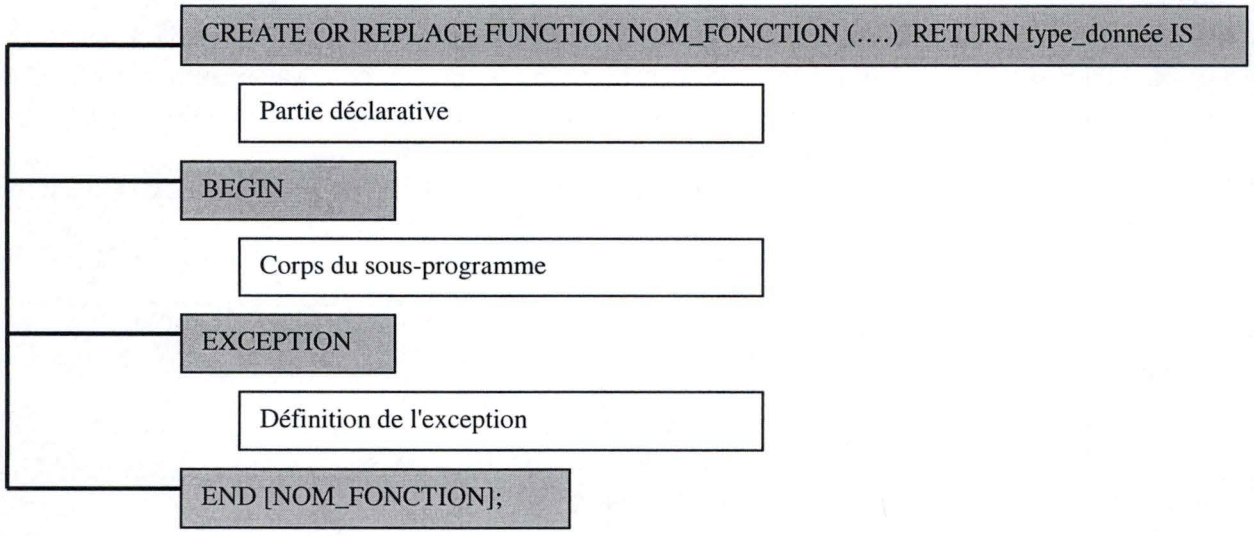


Schéma d'une fonction

Exemple de fonction : la fonction *demain* retourne la date du jour suivant la date du système. Cette fonction peut être utilisée pour définir la date de départ d'un placement (le lendemain du dépôt).

```
CREATE OR REPLACE FUNCTION demain RETURN DATE IS
BEGIN
RETURN (SYSDATE[6] +1);
END;
```

Les fonctions possèdent certains avantages par rapport aux procédures :

- La valeur de retour d'une fonction peut être un argument d'une procédure. En effet, une procédure dont le but est d'ajouter les placements financiers peut comprendre dans ces arguments la valeur *demain* pour indiquer la date de début du placement :
  ajout_placement (nom_client,….,montant, *demain*,..);

- Le résultat de la fonction peut, en fonction du type de la donnée, faire l'objet d'opérations. Puisqu'il est possible d'effectuer des opérations d'addition sur les variables de type date, pour un client privilégié, la date de début du placement peut être la date du jour du placement précédent le dépôt. L'instruction suivante affiche la date du début du placement.
  DBMS.OUTPUT.PUT_LINE (SYSDATE – 1);

- Les fonctions peuvent également être utilisées au sein de prédicats, ce qui permet de rendre le code plus lisible et plus facile à maintenir. Par exemple, les instructions suivantes permettent d'utiliser la fonction *demain* au sein d'un commande d'itération LOOP

```
WHILE variable_date < demain LOOP
   corps du loop;
END LOOP;
```

Ce simple exemple montre comment une fonction peut être utilisée au sein d'un condition d'arrêt d'une boucle. L'écriture de la condition est relativement claire à comprendre.

4.2.3 Les packages

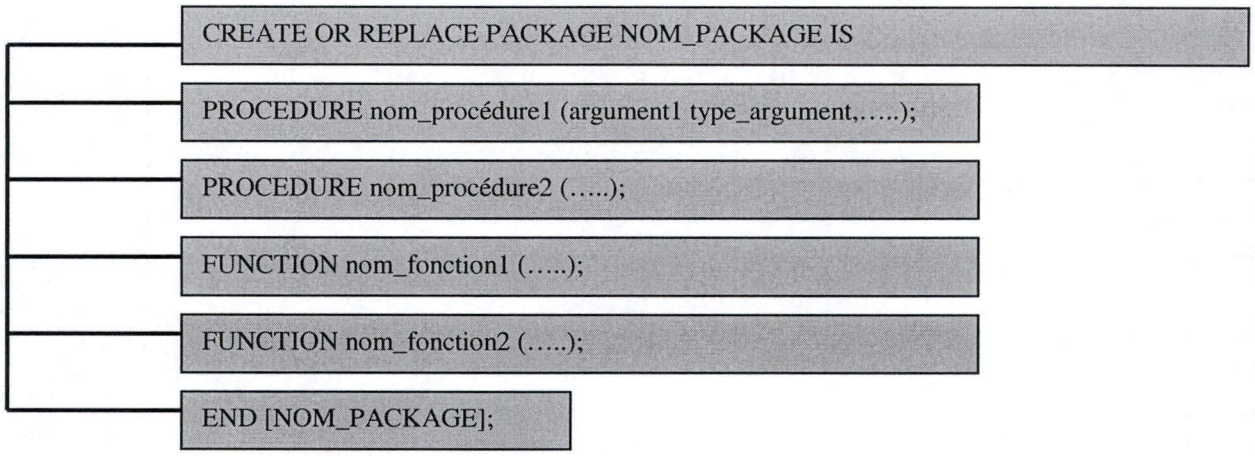
Les packages encapsulent des procédures et des fonctions reliées au sein d'une même librairie. En effet, chaque package contient un groupe de programmes compilés formant, de cette manière, une unique librairie d'objets.

Un package comprend deux parties :

- *La spécification du package* : toutes les constructions publiques du package sont déclarées. En résumé, la spécification permet de présenter ce qui peut être fait par les procédures ou les fonctions du package; elle présente la forme d'appel des objets (nom + arguments).
- *Le corps du package* : toutes les constructions spécifiées (publiques ou privées) sont définies. Il montre comment les procédures et les fonctions remplissent leur objectif.

[6] Fonction prédéfinie spécifiant la date courante du système.

Cette séparation permet une plus grande flexibilité dans le développement. De cette manière, on peut créer les spécifications et les références aux procédures ou fonctions déclarées publiques sans devoir développer le corps de chacun des éléments. En outre, on peut modifier le corps des procédures séparément de leur spécification déclarée. Ainsi, aussi longtemps que les spécifications des procédures ne sont pas modifiées, tout objet qui fait référence à ces éléments n'est pas considéré invalide. Schématisons les deux parties d'un package :



Spécification d'un package

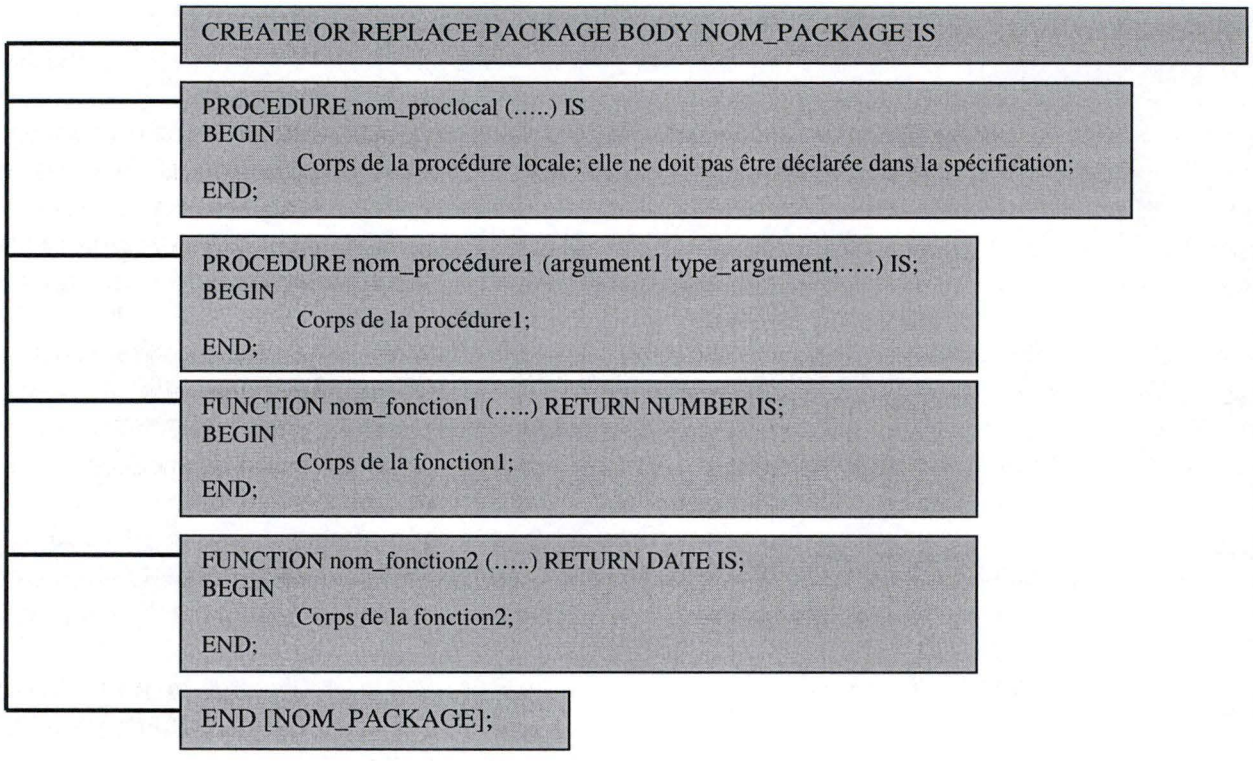


Diagramme du corps du package

Exemple de package : voici un package de nom "*gestion_clients*" regroupant trois procédures (2 procédures déclarées publiques, une procédure locale AJOUT_JOURNAL utilisée au sein de la procédure DEBIT_CREDIT_COMPTE et une fonction COMPTE_TRANS_JOUR).

CREATE PACKAGE gestion_clients **IS**
/* spécification du package gestion_clients */
solde_minimum CONSTANT NUMBER := 1000.00;
PROCEDURE AJOUT-CLIENT (nom VARCHAR2, prenom VARCHAR2, numbroker NUMBER; date DATE, compte NUMBER, ...);

```
PROCEDURE DEBIT_CREDIT_COMPTE (numcompte NUMBER, montant NUMBER);
FUNCTION COMPTE_TRANS_JOUR (date DATE) RETURN NUMBER;
END gestion_clients;
/* fin de la spécification du package gestion_clients */

CREATE PACKAGE BODY gestion_clients IS
/* début de la définition du corps des procédures et fonctions du package gestion_clients */
PROCEDURE AJOUT_JOURNAL(compte NUMBER, date DATE, montant NUMBER, utilisateur
VARCHAR) IS
/* procédure locale permettant d'insérer une transaction dans le livre-journal en ajoutant respectivement un
numéro de référence (incrémentation automatique), le numéro du compte, la date et le montant de l'opération*/
BEGIN
INSERT INTO JOURNAL VALUES
        (journal_ref.NEXVAL, compte, date, montant);
END;
PROCEDURE AJOUT_CLIENT( nom VARCHAR2, prenom VARCHAR2, numbroker NUMBER; date
DATE, compte NUMBER...) IS
BEGIN
INSERT INTO CLIENTS VALUES
        (client_numid.NEXVAL, nom, prenom, numbroker, date, compte..);
END;
PROCEDURE DEBIT_CREDIT_COMPTE (numcompte NUMBER, montant NUMBER) IS
/* Déclaration des variables /*
old.solde NUMBER;
new.solde NUMBER;
date DATE;
BEGIN
SELECT solde INTO old.solde FROM comptes
        WHERE compte_num=numcompte;
        FOR UPDATE OF solde;
        new.solde := old.solde + montant;
        IF (new.solde >= solde_minimum) THEN
        UPDATE comptes SET solde = new.solde WHERE compte_num=numcompte;
        AJOUT_JOURNAL(numcompte, SYSDATE, montant);
        COMMIT;
        ELSE
        DBMS_OUTPUT.PUT_LINE ('Opération annulée, votre compte ne peut être inférieur à 1000 francs);
        ROLLBACK;
        ENDIF
        EXCEPTION
        WHEN NO_DATA_FOUND and (montant >0) THEN
                INSERT INTO compte (compte_num, solde) VALUES (numcompte, montant);
        AJOUT_JOURNAL(numcompte, SYSDATE, montant);
        COMMIT;
        WHEN OTHERS THEN ROLLBACK;
END DEBIT_CREDIT_COMPTE;
CREATE FUNCTION COMPTE_TRANS_JOUR (date) RETURN NUMBER IS
compte_operation NUMBER;
BEGIN
        SELECT COUNT(*) AS compte_operation FROM JOURNAL WHERE date= date;
        RETURN (compte_operation);
END;
END gestion_clients;
```

Avantage du package :

- *Encapsulation* : les packages offrent la possibilité de grouper des procédures, des fonctions et des types de données en une seule "unit"; ce qui offre une meilleure organisation durant la phase de développement.

- *La déclaration des éléments "privés ou* publics". Les éléments déclarés "publics" sont accessibles directement à tous utilisateurs du package. Les parties privées sont à usage interne au package et donc ne peuvent être utilisées extérieurement au package.

- *L'utilisation des packages accroît la performance* : lorsqu'une procédure d'un package est appelée pour la première fois, le package en entier est enregistré en mémoire. Par conséquent, tout appel supplémentaire d'une autre procédure ou fonction du package n'exige aucune commande d'entrée ou sortie pour exécuter le code en mémoire. Le corps du package peut être remplacé et recompilé sans affecter la partie des spécifications.

4.2.4 Les packages prédéfinis

Dans tout système, est fournie une série de packages prédéfinis offrant aux utilisateurs, une série de procédures et de fonctions compilées directement opérationnelles. Oracle ne déroge pas à la règle. Citons quelques exemples de packages prédéfinis. La liste qui suit n'est pas exhaustive.

- STANDARD : ce package reprend un ensemble de fonctions
  - numériques : ABS, SQRT, TRUNC, ...
  - de traitement du type "caractère" : CONCAT, INITAP, LOWER, UPPER,...
  - de conversion : TO_CHAR, TO_DATE, TO_NUMBER,...
  - de traitement du type "date" : SYSDATE, NEXT_MONTH,...
  - ...
- DBMS_TRANSACTION : ce package globalise les fonctions relatives à l'exécution de transactions : SET TRANSACTION, COMMIT, SAVEPOINT, ROLLBACK...
- DBMS_OUTPUT : ce package regroupe un ensemble de fonctions relatives à la gestion des buffers d'affichage : PUT_LINE, GET_LINE, GET_LINES...
- DBMS_PIPE et DBMS_ALERT : ces deux packages rassemblent les fonctions implémentant les mécanismes de communication inter-processus qui sont, respectivement, similaires aux pipes et signaux de UNIX.

4.3 Avantages des procédures stockées

4.3.1 Sécurité

Les principes de sécurité de données, qui sont actuellement un objectif primordial au sein des SGBD, peuvent être mis en vigueur à travers les procédures stockées. En effet, il est tout à fait possible de réduire les opérations pouvant être appliquées sur la base de données en permettant aux utilisateurs d'accéder à l'information de la base, uniquement par les procédures ou fonctions prédéfinies. Par cette technique, les procédures et les fonctions définissent le cadre des opérations permises sur la base. Il est donc impossible de faire des mises à jour

directement dans une table puisque l'utilisateur ne dispose pas des droits d'accès pour la manipuler. Les utilisateurs qui n'ont que les privilèges d'exécution des fonctions ou des procédures, ne peuvent manipuler les données d'aucune autre manière que celle imposée par le programmeur. Cela permet de cerner le cadre de traitement de chaque utilisateur en fonction de leurs responsabilités.

4.3.2 Performance et productivité

L'ensemble des procédures stockées et compilées est directement disponible dans la base de données. Il n'est donc pas nécessaire d'effectuer une compilation de ces éléments à chacune de leur exécution.
Sur un serveur, les procédures stockées ont la caractéristique que le volume d'informations diffusées à travers le réseau est nettement plus faible comparé au volume propagé par les requêtes SQL ou par un bloc PL/SQL puisque l'information est envoyée une et une seule fois et appelée ensuite lorsqu'elle est utilisée.

Il existe encore certains avantages de performance liés aux procédures stockées propres à certains SGBD. Oracle qui utilise le SGA (System Global Area[7]) offre l'avantage suivant : si la procédure est déjà présente dans cet ensemble, l'exécution peut s'enclencher directement. De même, une seule copie des procédures ou des fonctions doit être enregistrée en mémoire pour un usage multi-utilisateurs.

Les procédures stockées augmentent la productivité des développements. En effet, en développant des applications basées sur un ensemble de procédures communes, cette technique permet d'éviter la réécriture de certaines parties et donc une multiplicité de redondances. En outre, la maintenance est rendue plus aisée puisque toute modification se fait au niveau des procédures ou des fonctions. Les changements sont directement applicables dans chacune des applications qui font appel à ces éléments.
De plus, lorsque les applications font référence à un ensemble commun de procédures (compilées et testées), il est évident que cela réduit les erreurs de développement puisque le code offre une meilleure lisibilité. Les procédures utilisées ne doivent plus être vérifiées et peuvent être exécutées autant de fois que possible sans devoir être recompilées.

4.4 Exécution des procédures stockées

Avant toute exécution d'une procédure, le SGBD vérifie :

- *Le droit d'accès de l'utilisateur* : le système vérifie que l'utilisateur appelant la procédure est, soit le propriétaire, soit possède les droits d'exécution sur l'objet invoqué.

- *La validité de la procédure* : le SGBD évalue le statut de la procédure car il ne peut exécuter une procédure invalide. Une procédure est déclarée invalide si

    - Un objet référencé (table, colonne, procédure, fonction, variable,..) de la procédure a été modifié ou supprimé;
    - Les privilèges d'utilisation ont été supprimés;
    - La procédure appelée n'est pas déclarée publique.

[7] Il s'agit d'un ensemble de mémoires partagées comprenant tous les verrous, les buffers de la base, le dictionnaire des données et encore bien d'autres éléments.

Si une procédure invalide est appelée, le système effectuera, avant de générer une erreur, une nouvelle compilation de l'élément défaillant afin de vérifier que le problème n'a pas été résolu entre-temps.

Les deux vérifications opérées et satisfaites permettent l'exécution de la procédure, de la fonction ou du package selon les étapes suivantes :
Une procédure stockée peut être appelée depuis un shell SQL comme SQL*PLUS d'Oracle, depuis tout programme d'applications (C, Powerbuilder, JAVA..) mais également depuis une applet java; ce qui permet une intégration dans le web.

- *Exécution depuis* SQL*PLUS *:*

Une fois les packages compilés, il est possible d'exécuter toutes les procédures ou fonctions créées. En supposant que nous sommes connectés via une session SQL*Plus, l'exécution de la fonction *demain* est illustrée par les lignes suivantes :

```
SQL > VARIABLE date_demain DATE     Déclaration de la variable
SQL > EXECUTE date_demain := demain; Assignation de la variable
SQL > PRINT date_demain;          Affichage de la variable
14-JUN-99
SQL >
```

- *Exécution depuis un programme C :*

Supposons une procédure stockée "affichage_client" du package *gestion* dont le but est d'afficher les informations du dernier client encodé de la table *CLIENTS*. Cette procédure (sans argument) peut être appelée par un programme C.

```
Main ()
{       EXEC SQL BEGIN DECLARE SECTION
        char infoclient[255];
        EXEC SQL END DECLARE SECTION;
        EXEC SQL EXECUTE
                BEGIN
                        gestion.affichage_client;
                        DBMS_OUTPUT.GET_LINES(:infoclient);
                END;
        Printf ("%s\n", infoclient);
}
```

Ce programme exécute la procédure stockée, récupère le contenu du buffer d'affichage DBMS_OUTPUT par l'exécution de la procédure stockée GET_LINES et enfin, affiche le résultat sur son propre buffer par la commande printf. Pour pouvoir effectuer ces opérations, le programme d'application doit disposer des droits d'exécution sur le package ainsi que des droits d'accès pour pouvoir manipuler la table concernée.

Ayant précisé les caractéristiques des procédures stockées, notre étude va analyser un type spécifique de procédures stockées : les triggers. Ils permettent d'implémenter, au sein des SGBDA, les principes des règles actives que nous avons dégagés au cours des chapitres précédents.

Chapitre 5:

Les Triggers

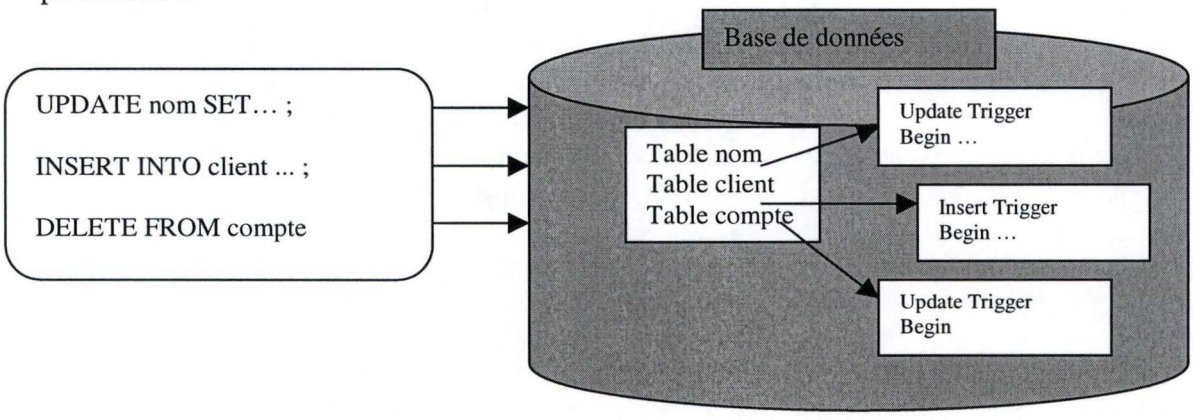
Chapitre 5 :

Les Triggers

5.1 Définition des triggers

Les triggers (déclencheurs) sont des procédures stockées exécutées implicitement par un SGBD en réponse à l'exécution d'une instruction INSERT, UPDATE ou DELETE. Ils se composent d'instructions SQL, PL/SQL qui s'exécutent en un seul bloc. Ils peuvent, au cours de leur traitement, opérer différentes manipulations suivant les règles établies et, en cas de problèmes, générer des erreurs (RAISE_APPLICATION_ERROR[1]). Lorsqu'une exception est générée par un trigger ou lorsqu'une erreur d'exécution se produit, les effets des instructions sont automatiquement annulés. En effet, les triggers effectuent des traitements possédant les caractéristiques classiques des transactions :

- *Atomicité* : le contenu logique est exécuté complètement ou pas du tout;
- *Cohérence* : si la base est cohérente avant l'action, elle le sera après également;
- *Indépendance* : le traitement est effectué indépendamment des autres traitements en cours;
- *Durabilité* : si la transaction se clôture correctement alors la mise à jour est permanente.



Déclaration des triggers au sein de la base de données

Les triggers et les procédures stockées se distinguent par la manière dont ils sont appelés. En effet, une procédure est exécutée explicitement par un utilisateur, une application ou un trigger. Les triggers sont, quant à eux, implicitement exécutés par le SGBD lorsque les conditions de déclenchement sont constatées. Il n'y a dès lors aucune intervention de la part d'un quelconque utilisateur ou d'une application.

Les triggers peuvent fournir, en supplément des caractéristiques standards des systèmes de gestion de base de données, des systèmes hautement personnalisés en fonction des besoins

[1] RAISE_APPLICATION_ERROR est une procédure stockée prédéfinie qui a pour effet de remplir les champs d'une structure SQLCA qui est accessible aux programmes d'application effectuant des requêtes sur la base de données. Ce mécanisme de gestion d'erreurs a notamment été mis en place parce que les applications sont incapables de gérer des exceptions Oracle propagées jusqu'à elles.

spécifiés. Un trigger peut, par exemple, réduire les opérations Data Management Language[2] sur une table en fonction des heures de travail. Il est également possible de déclencher certaines opérations durant des moments prédéterminés (calcul de consolidation de compte à 23h00; backup de certaines données à 00h00;...).

Les références de ce chapitre sont :

- Owens, Kevin T.,"*Building Intelligent Databases with Oracle PL/SQL, Triggers, & Stored Procedures*", Prentice Hall, 1998.
- Aide en ligne d'Oracle, "*Concept Release8, DataBase Triggers*".
- http://www.oracle.com
- http://www.orafans.com/ :The Oracle User Forum
- Delmal Pierre, "*SQL2- application à Oracle, Access et RDB*", De Boeck Université, 1998.

5.2 Utilisation des triggers

Il existe de nombreux exemples de triggers. A titre documentaire, nous pouvons citer différentes formes d'applications utilisant la logique des triggers dans la réalité :

1) Un trigger, déclenché par une instruction d'insertion ou de mise à jour, examine la validation des données provenant d'une base de données éloignée. Le trigger génère une exception si les conditions de validation ne sont pas vérifiées.

2) Un trigger qui examine les limites de certaines valeurs lors d'une instruction d'insertion ou de mise à jour. En effet, si certaines valeurs de colonne excèdent les limitations imposées, le trigger les remplace par une valeur par défaut.

3) Un trigger qui analyse des données nouvellement introduites par une instruction d'insertion et qui les envoie vers un autre traitement via une instruction DBMS[3].

4) Un trigger qui envoie un message d'alerte vers un autre traitement, chaque fois que des valeurs agrégées au sein de la base de données atteignent ou dépassent les limites imposées.

5) Un trigger qui se déclenche lorsqu'une machine qui se connecte à un processus industriel avertit le responsable que certains réglages doivent être effectués...

Ces quelques exemples reflètent les implémentations des triggers. Cependant, on peut imaginer des triggers qui se déclencheraient lors d'autres évènements externes à la base de données : Citons l'envoi d'un message depuis un programme applicatif ou encore, l'envoi d'un message en fonction des privilèges des utilisateurs, des heures de traitement..

Comme nous venons de le voir, l'utilisation des triggers est très variée. Nous pouvons énoncer les principales formes d'utilisation des triggers dans les applications de base de données les plus courantes. Leur but est de :

---

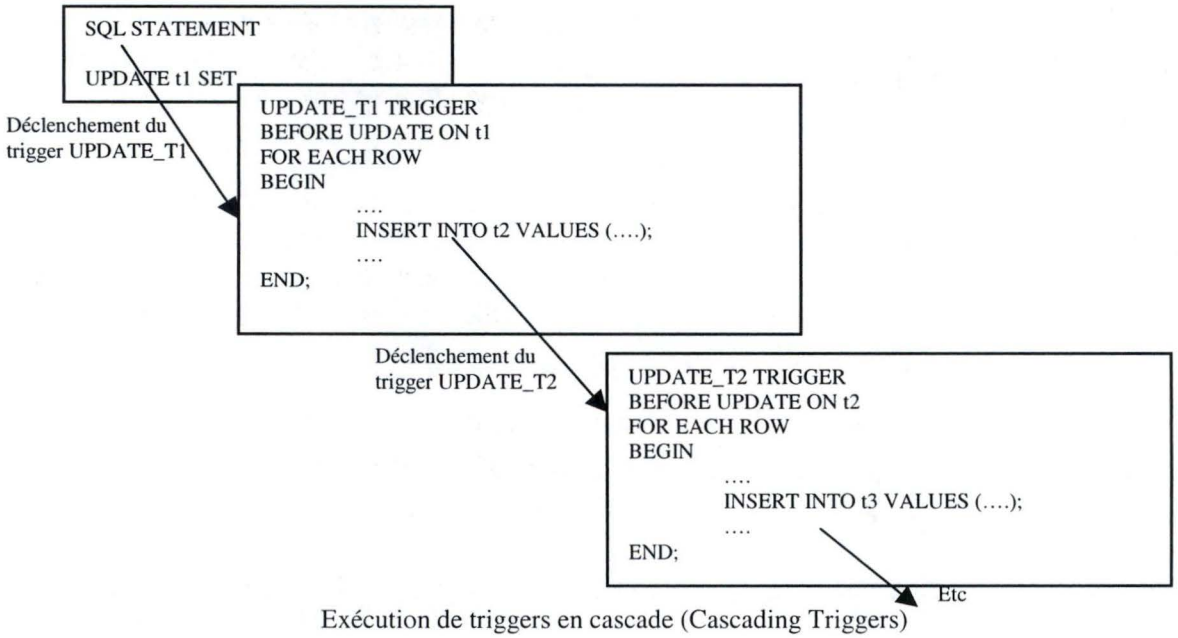
[2] Pour rappel, les commande DML (Data Manipulation Language) manipulent de l'information au sein d'une base de données à travers les commandes de lecture (SELECT) , d'insertion (INSERT), de mise à jour (UPDATE) et de suppression (DELETE). En outre, ce sont les commandes DDL (Data Definition Language) qui permettent de créer les objets de la base de données (tables, contraintes, procédures stockées et triggers).

[3] Database management system

- Générer automatiquement la valeur de certaines colonnes en fonction de règles bien précises;
- Éviter des transactions invalides;
- Faire respecter des contraintes de sécurité prédéfinies : aucun encodage n'est permis entre 18h00 et 9h00;
- Imposer les contraintes d'intégrité référentielle émises au sein de la base de données distribuées : les clients américains effectuant des opérations en Belgique doivent être encodés dans la filiale américaine;
- Appliquer des règles complexes émises par la direction : seul le directeur financier (utilisateur : Dupond) peut effectuer des calculs de consolidation; uniquement entre le 25 et la fin de chaque mois;
- Fournir de l'information sur les connexions : quelles sont les personnes qui ont effectué des transactions durant la dernière semaine;
- Effectuer des audits complexes;
- Assurer la duplication des tables temporelles ;
- Distribuer des statistiques sur l'accès aux tables ...

Comme nous pouvons le constater, l'utilisation des triggers peut être stratégique car des règles complexes d'intégrité, de sécurité et d'analyse peuvent être générées par ces techniques.

Les triggers sont d'une grande utilité lorsque l'on désire personnaliser et dynamiser une base de données. Cependant, il est fortement conseillé de les insérer uniquement dans les cas nécessaires. En effet, leur utilisation et les mises en pratique peuvent poser de multiples problèmes si leur maîtrise n'est pas contrôlée. En effet, lorsque l'action d'un trigger contenant des instructions INSERT, UPDATE ou DELETE est exécutée, d'autres triggers, à leur tour, peuvent se déclencher en cascade. En effet, au cours du traitement d'un trigger, d'autres instructions peuvent déclencher conjointement d'autres triggers (triggers en cascade). Ces enchaînements en cascade peuvent ralentir le système et même engendrer des phénomènes de bouclage. Cependant, le système prévoit des techniques pour éviter et contrecarrer ce bouclage en limitant le nombre d'appels des triggers en cascade.
Un trigger peut en déclencher d'autres:



Exécution de triggers en cascade (Cascading Triggers)

Cette pratique peut poser de nombreux problèmes de performance. L'utilisation excessive de triggers peut être une conséquence logique dans les cas d'interdépendances complexes. Nous reviendrons plus en détails sur leur application.

5.3 Le triggers et le modèles ECA

Le mécanisme des triggers implémente le modèle Evénement-Condition-Action : en fonction de la survenance d'un événement (événements d'insertion, de mise à jour ou de suppression dans la base), et si une condition est vérifiée, alors un trigger se déclenche et exécute les actions définies.

Un trigger est donc constitué de trois parties bien distinctes.

1. A triggering event or statement : **EVENEMENT**

2. A triggering restriction : **CONDITION**

3. A trigger action : **ACTION**

Nous pouvons examiner plus en détails ces trois parties à travers un simple exemple d'un trigger PL/SQL qui déclenche la commande automatique d'un produit lorsque son niveau de stockage est en deçà d'une limite fixée.

```
AFTER UPDATE OF niveau_stock ON inventaire                    TRIGGERING STATEMENT
WHEN (new.niveau_stock<new.limite_commande_auto)              TRIGGER RESTRICTION
FOR EACH ROW
DECLARE
NUMBER compteur ;
BEGIN
        SELECT COUNT(*) INTO compteur
        FROM commande_cours
        WHERE renouvellement_num= : new . renouvell ment_num ;
        IF compteur =0
        THEN                                                  TRIGGER ACTION
                INSERTS INTO commande_co rs
                VALUES (new . renouvellement num , new . renouvellement_quantite , sysdate) ;
        ENDIF ;
END ;
```

5.3.1 Triggering event and statement

Cette première partie comprend uniquement une instruction SQL qui est l'élément déclencheur du trigger. Celui-ci peut être une instruction d'insertion, de mise à jour ou de suppression portant sur une table de la base de données.

« **AFTER UPDATE OF** niveau_stock **ON** inventaire » signifie que, lorsque la colonne niveau_stock d'une ligne de la table *inventaire* est mise à jour, le trigger est automatiquement déclenché. C'est, par conséquent, la conjonction du type d'instruction (INSERT, DELETE ou UPDATE) et de la liste des colonnes impliquées qui permettent d'activer le trigger. L'événement déclencheur

peut être une combinaison d'instructions (Data Manipulation Language statement)[4]. Par exemple, « INSERT OR UPDATE OR DELETE ON inventaire » permet de déclencher le trigger pour toute insertion, mise à jour ou suppression sur la table *inventaire*. De cette manière, il est possible de créer un seul trigger qui exécute une certaine action bien précise en fonction du mode de déclenchement.

5.3.2 Trigger restriction

Cette partie est une simple expression logique (booléenne) qui doit être vraie pour exécuter le trigger. L'action associée au trigger ne sera en aucun cas exécutée si la condition est jugée fausse ou inconnue. Cette partie est optionnelle.

La commande WHEN (new.niveau_stock<new.limite_commande_auto) évalue la condition de déclenchement du trigger. Le trigger se déclenchera et effectuera l'action associée si, lors d'une opération de mise à jour sur le niveau de stock de la table *inventaire*, une valeur modifiée de la colonne niveau_stock est inférieure à la limite de stockage. En cas d'évaluation positive (TRUE), la procédure de commande automatique se lance (Trigger Action). Il est tout à fait possible de transférer le test logique de l'opération WHEN dans le corps du trigger.

5.3.3 Trigger action

L'action qui est une simple procédure (un bloc d'instructions SQL, PL/SQL), est réalisée lorsque les conditions d'exécution sont réunies ; c'est-à-dire lorsque l'événement a eu lieu et que les conditions de réalisation sont jugées vraies. Comme les procédures stockées, l'action du trigger est un ensemble de codes procéduraux SQL et PL/SQL permettant d'effectuer les traitements désirés. Notre exemple permet d'insérer dans la table des commandes en cours "commande_cours", une nouvelle commande associée au produit concerné si cette mesure n'a pas été opérée au préalable. Cette commande est composée de trois champs : le numéro du produit concerné, la quantité devant être commandée et la date de commande. Notons que le SGBD conserve, sur la durée d'exécution d'une mise à jour, deux variables contenant respectivement l'ancienne et la nouvelle valeur de la colonne modifiée. Ce sont les valeurs de corrélation relatives à la modification[5].

5.4 Types de triggers

Oracle classifie les triggers selon trois typologies.

5.4.1 Action

Cette typologie classifie les triggers en fonction de l'événement déclencheur associé (triggering statement). Il s'agit de l'exécution de commandes INSERT, UPDATE ou DELETE.

---

[4] Oracle sépare les commandes des bases de données en six catégories. Parmi celles-ci, les commandes DDL (Data Definition Language) permettent de créer les objets suivants : tables, contraintes déclaratives, procédures stockées et triggers. Ces objets sont modifiés par des commandes DDL. Les commandes DML (data Manipulation language) effectuent des requêtes sur la base de données (SELECT) et effectuent des modifications sur celle-ci à travers les instructions DELETE, INSERT et UPDATE.

[5] Le concept des valeurs de corrélation est défini P.59.

5.4.2 Level

Lorsque l'on définit un trigger, on peut moduler le nombre de fois que le trigger devra s'exécuter. S'exécutera-t-il une seule fois au cours du traitement (au niveau global de l'instruction) ou à chaque modification d'un enregistrement (au niveau des enregistrements)? En effet, la seconde typologie permet de distinguer deux types de triggers en fonction du nombre de déclenchements : il s'agit des triggers "statement-level" et les triggers"row-level".

- **Les triggers "statement-level"**

  Ces triggers sont exécutés une seule fois par exécution d'une instruction INSERT ,UPDATE ou DELETE et cela quel que soit le nombre d'enregistrements modifiés. Ce type de trigger est utilisé dans le cas où le code d'action associée est indépendant de l'instruction de déclenchement ou des données mises à jour. On utilise généralement ce type de triggers, soit pour implémenter des traitements portant sur plusieurs lignes de tables, soit pour opérer une vérification sur la personne connectée, sur l'heure de connexion ou, encore, pour opérer un audit particulier sur un ensemble de données selon un type particulier de déclenchement. Citons quelques modélisations de ce type de triggers :

  - La somme des actions achetées ne peut dépasser la somme des montants des comptes clients.
  - En dehors des heures ouvrables, seuls les membres appartenant au Front Office peuvent effectuer des opérations de suppression de comptes.
  - Un client ne peut acheter plus de 10 millions de produits financiers par jour sans remplir un formulaire spécifique pour la Commission bancaire...

- **Les triggers "row-level"**

  Ils sont exécutés à chaque modification d'un enregistrement de la table concernée. Lorsqu'une instruction de mise à jour modifie plusieurs lignes d'une table, le trigger row-level s'exécutera à chaque mise à jour d'un enregistrement. Si l'instruction UPDATE n'affecte aucune ligne, le trigger ne sera pas exécuté.
  Ce trigger est aisément identifiable puisque, dans le corps du trigger, se trouve le code "FOR EACH ROW".

  Ce type de trigger est utilisé lorsque le code procédural de l'action associée dépend des données fournies par l'instruction "déclencheur". Supposons une table dans laquelle 25 lignes doivent être modifiées et un trigger row-level défini sur cette table pour une opération de mise à jour, une instruction UPDATE sur cette table provoquera 25 exécutions du trigger.

  En résumé, un trigger row-level est utilisé pour vérifier une contrainte d'intégrité ou tout autre traitement dont la portée est limitée à un seul enregistrement. Citons différents exemples :

  - Une commande comporte 10% de frais divers.
  - Les taux de change proposés aux clients ne peuvent fluctuer de plus de 5% (sauf en cas de dévaluation).

- Un compte client ne peut être supprimé si des ordres permanents sont encore encodés.

On aurait pu imaginer des triggers de type WHILE, qui se déclencheraient pendant l'exécution d'une instruction INSERT, UPDATE ou DELETE. Ce trigger hypothétique pourrait bloquer l'exécution d'une instruction DML et transformer l'action par son propre code. En effet, si l'on voulait forcer l'encodage de données selon un horaire déterminé, on pourrait écrire un trigger qui empêcherait tout encodage en dehors de plages horaires en comparant la date du système, l'heure d'encodage et les plages permises. Le trigger pourrait annuler complètement l'exécution de l'instruction en la remplaçant par sa propre exécution (afficher un message, d'une part, à l'utilisateur ne respectant pas la règle et, et d'autre part, à la sécurité, pour analyse complémentaire).

5.4.3 Timing

Oracle distingue les triggers qui s'exécutent avant les instructions d'INSERT, UPDATE ou DELETE – triggers de type BEFORE - de ceux qui s'exécutent après, -triggers de type AFTER.

Puisque les valeurs de corrélation sont disponibles durant toute la séquence d'exécution d'un UPDATE, INSERT ou DELETE, les triggers de type BEFORE et AFTER peuvent y accéder sans problème. Cependant, la différence entre ces deux types de triggers réside dans le fait que les triggers BEFORE peuvent exiger l'écriture d'une valeur de NEW.x[6] qui sera ensuite inscrite dans la table correspondante. Cette valeur pourrait être différente de celle qui aurait été écrite par l'instruction INSERT ou UPDATE.

5.5 Douze triggers possibles

Il est possible d'associer douze triggers à chaque table d'une base de données; ceux-ci se distinguant par leur mode de déclenchement. Ces triggers possèdent des attributs définis dans les trois parties définies précédemment : Action (instruction SQL d'insertion, de mise à jour ou d'insertion), Level (statement ou row-level) et Timing (Before ou After).

Résumons les 12 triggers[7] possibles :

Insert Triggers	Update Triggers	Delete Triggers
Before Insert Statement	Before Update Statement	Before Delete Statement
Before Insert Row	Before Update Row	Before Delete Row
After Insert Statement	After Update Statement	After Delete Statement
After Insert Row	After Update Row	After Delete Row

Résumons brièvement les différents types de triggers (code PL/SQL) d'une table clients en fonction de certaines conventions d'écriture exposées par la suite.

---

[6] cfr 5.7 les valeurs de corrélation.

[7] Depuis la release 7.3, Oracle permet de définir plusieurs types de triggers sur une table. C'est ainsi que, par exemple, plusieurs triggers Before Insert Statement peuvent être associés à une table. Différents outils CASE peuvent faciliter la logique de programmation de ces triggers.

Type de trigger	Nom du trigger	Code PL/SQL de déclenchement

Before Insert Statement	CLI_BIS	CREATE OR REPLACE TRIGGER CLI_BIS
		BEFORE INSERT ON CLIENTS
		BEGIN
		Body
		END;

Before Insert Row	CLI_BIR	CREATE OR REPLACE TRIGGER CLI_BIR
		BEFORE INSERT ON CLIENTS
		FOR EACH ROW
		BEGIN
		Body
		END;

After Insert Row	CLI_AIR	CREATE OR REPLACE TRIGGER CLI_AIR AFTER INSERT ON CLIENTS FOR EACH ROW BEGIN *Body* END;.

After Insert Statement	CLI_AIS	CREATE OR REPLACE TRIGGER CLI_AIS
		AFTER INSERT ON CLIENTS
		BEGIN
		Body
		END;

Before Update Statement	CLI_BUS	CREATE OR REPLACE TRIGGER CLI_BUS
		BEFORE UPDATE ON CLIENTS
		BEGIN
		Body
		END;

Before Update Row	CLI_BUR	CREATE OR REPLACE TRIGGER CLI_BUR
		BEFORE UPDATE ON CLIENTS
		FOR EACH ROW
		BEGIN
		Body
		END;

After Update Row	CLI_AUR	CREATE OR REPLACE TRIGGER CLI_AUR
		AFTER UPDATE ON CLIENTS
		FOR EACH ROW
		BEGIN
		Body
		END;

After Update Statement	CLI_AUS	CREATE OR REPLACE TRIGGER CLI_AUS
		AFTER UPDATE ON CLIENTS
		BEGIN
		Body
		END;

Before Delete Statement	CLI_BDS	CREATE OR REPLACE TRIGGER CLI_BDS
		BEFORE DELETE ON CLIENTS
		BEGIN
		Body
		END;

Before Delete Row	CLI_BDR	CREATE OR REPLACE TRIGGER CLI_BDR
		BEFORE DELETE ON CLIENTS
		FOR EACH ROW
		BEGIN
		Body
		END;

After Delete Row	CLI_ADR	CREATE OR REPLACE TRIGGER CLI_ADR
		AFTER DELETE ON CLIENTS
		FOR EACH ROW
		BEGIN
		Body
		END;

After Delete Statement	CLI_ADS	CREATE OR REPLACE TRIGGER CLI_ADS
		AFTER DELETE ON CLIENTS
		BEGIN
		Body
		END;

5.6 Identification des triggers

Le nom des triggers doit être unique lors de leur définition. Les noms des triggers peuvent fonctionner avec d'autres noms d'objets (procédures ou tables par exemple). Cependant, il faut garder à l'esprit que, si une table, une procédure et un trigger portent le même nom, une certaine confusion peut apparaître. La définition des noms des triggers peut être générique et pas simplement découler d'une fonction spécifique.

Supposons un trigger « VERIF_MONTANT_LIMIT » qui se déclenche lors de la mise à jour d'un montant de la table TRANSACTIONS. Si le code d'action de ce trigger est modifié par une exigence particulière indépendante du montant. Le nom du trigger n'est plus directement lié aux actions accomplies. Si une erreur intervient, Oracle peut générer une erreur de type :

ORA-08500 :PL/SQL : numeric or value error
ORA-06056 : error during execution of trigger 'VERIF_MONTANT_LIMIT '

Ce genre de message ne précise nullement si le trigger s'est déclenché suite à une instruction de mise à jour, de suppression ou d'insertion. Il n'identifie pas la table concernée.

Pour éviter ces problèmes, des conventions de dénomination des triggers ont été établies :

Conventions de dénomination des triggers :

Nom_du_trigger = Nom_de_la_ table_[A | B]_[I | U | D] _[S | R]

Où

[A | B] désigne une After ou Before Trigger;

[I | U | D] désigne un trigger associé à une instruction Insert, Update ou Delete;

[S | R] Désigne un "Row ou Statement Level" Trigger.

Par ces conventions, l'identification d'un trigger impliqué dans une erreur est facilitée.

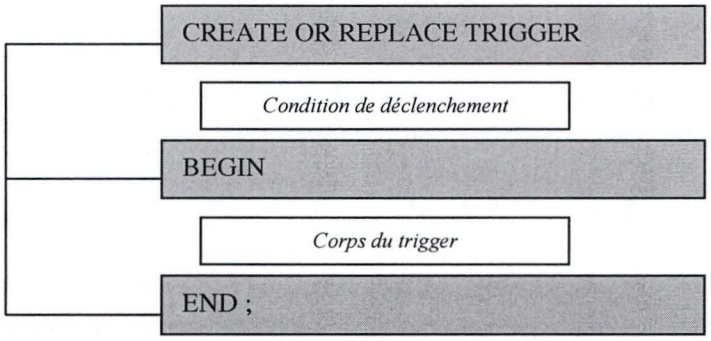
ORA-08500 : PL/SQL : numeric or value error
ORA-06056 : error during execution of trigger "TRANSACTIONS_AIR"

Ce message permet de localiser directement le problème sur le code du trigger after-insert-row de la table Transactions.

5.7 Déclaration et description des triggers PL/SQL

Un trigger est donc une procédure stockée sans aucun paramètre, appelée au sein d'une base de données. Défini par la commande "CREATE or REPLACE TRIGGER", il est toujours associé **à une et une seule table** d'une base de données. Cela n'empêche qu'il peut exécuter des actions qui portent sur plusieurs tables. La première partie décrit les spécifications du déclenchement du trigger; c'est-à-dire quelle(s) instruction(s) et sous quelle(s) condition(s), le trigger est-il déclenché. Le corps du trigger est simplement un bloc PL/SQL regroupant l'ensemble des instructions à accomplir en réponse conjointe au déclenchement du trigger et au respect des conditions de réalisation.

Outre les instructions classiques PL /SQL, il peut se composer de variables, de constantes et de types de variables définies localement.



Définition d'un trigger Oracle PL/SQL

La syntaxe complète d'écriture d'un trigger PL/SQL est la suivante :

CREATE [OR REPLACE] TRIGGER nom_trigger
(BEFORE | AFTER)[1] (INSERT | UPDATE | DELETE) [1-3] **of** (nom_colonne[,nom_colonne...])[*]
on nom_table
[FOR EACH ROW][rl]
[**WHEN** expression_booléenne][cs]
BEGIN
Corps du trigger;
END;

[1] : élément obligatoire
[1-3] : un trigger peut être défini sur les trois opérations UPDATE, DELETE, INSERT.
[*] : uniquement pour les opérations de mise à jour (UPDATE), le trigger peut être défini sur une ou plusieurs colonnes. Cette partie est facultative.
[rl] : uniquement pour un trigger "Row Level"
[cs] : représente une option permettant de définir une condition supplémentaire de déclenchement. Le corps du trigger ne sera exécuté que dans le cas où l'expression booléenne est évaluée à *True*. Les valeurs de corrélation peuvent être utilisées à ce niveau pour décrire la condition.

Remarquons encore que les contrôles de transaction ne sont pas permis au sein des triggers. Des commandes telles que ROLLBACK, COMMIT et SAVEPOINT[8] qui seraient définies au sein d'un trigger entraîneraient directement des erreurs de compilation. Notons encore que les procédures stockées qui peuvent être appelées au sein d'un trigger, ne peuvent en aucun cas utiliser les commandes de transaction sous peine de générer également une erreur et entraîner l'annulation complète de l'opération. La méthode conseillée pour effectuer ces contrôles de transaction est de faire appel au procédure RAISE_APPLICATION_ERROR du package prédéfini DBMS_STANDARD. Lorsque cette procédure est appelée, toutes les transactions associées sont annulées.

Les triggers peuvent être activés (Enable) et désactivés (Disable). Il est possible de désactiver et d'activer temporairement tous les triggers associés à une table (CLIENTS) par la commande suivante : ALTER TRIGGER CLIENTS DISABLE [ENABLE] ALL TRIGGERS; Pour effectuer la même opération sur un trigger particulier, il suffit d'exécuter la commande :

ALTER TRIGGER CLI_AUR DISABLE [ENABLE];

[8] Les opérations de transactions ROLLBACK, COMMIT et SAVEPOINT se définissent comme suit:
Rollback (Défaire) : opération d'annulation d'une transaction entraînant la suppression de tous ses effets.
Commit (Confirmation) : opération d'acceptation de la transaction. Cette commande permet l'écriture dans la base de données des changements opérés par la transaction.
Savepoint : commande permettant un point de sauvegarde intermédiaire au cours d'une transaction.

5.7 Les valeurs de corrélation

Comme nous l'avons déjà mentionné, les triggers "Statement Level" se déclenchent une seule fois au cours de l'instruction. Ce type de trigger ne donne, dès lors, aucune information sur les changements effectués et le nombre d'enregistrements modifiés. Les triggers "Row Level", quant à eux, s'exécute pour chacun des enregistrements affectés par l'instruction SQL. Il est par conséquent possible, au cours de l'instruction SQL, de posséder des informations sur chacun des enregistrements aussi bien avant qu'après les modifications. Pour marquer cette distinction, chaque valeur d'un enregistrement est marquée d'un préfixe "old" et "new" pour la valeur respectivement avant et après modification. Ces éléments sont appelés "les valeurs de corrélation".
Par exemple, le SGBD conserve, durant toute la durée d'exécution d'un UPDATE du champ TRA_QUANTITE sur un enregistrement quelconque de la table transaction, deux variables contenant respectivement l'ancienne (OLD.TRA_QUANTITE) et la nouvelle valeur (NEW.TRA_QUANTITE) de la colonne à modifier. Seuls les triggers "Row Level" ont accès à ces variables. Notons que, pour une instruction INSERT, la valeur de OLD. TRA_QUANTITE est NULL et en cas de suppression (DELETE), la valeur de NEW. TRA_QUANTITE est NULL.

Puisque les valeurs de corrélation sont disponibles sur toute la durée de l'exécution d'un UPDATE, INSERT ou DELETE, les triggers de type BEFORE, autant que ceux de type de type AFTER, peuvent y accéder. La différence entre ces deux types de triggers réside dans le fait que ceux de type BEFORE peuvent forcer directement l'écriture d'une valeur NEW.x qui est inscrite dans la colonne correspondante et qui est différente de la valeur qui aurait été écrite par une instruction d'insertion ou de mise à jour. Par exemple, un trigger de type BEFORE, déclaré sur la table TRANSACTIONS pour une opération de mise à jour peut, alors qu'une instruction UPDATE tente de modifier le montant (TRA_MONTANT) d'une transaction déjà introduite pour un broker[9] particulier à un niveau supérieure à 10.000.000 francs, assigner à NEW.TRA_MONTANT une valeur qui est différente de celle que l'UPDATE tentait d'écrire et qui correspond à la valeur de plafond de 10.000.000 de francs; l'ancienne valeur étant oubliée. C'est donc ce montant qui sera enregistré dans la base de données.

```
CREATE OR REPLACE TRIGGER TRANSACTIONS_BUR
BEFORE UPDATE OF TRA_MONTANT ON TRANSACTIONS
FOR EACH ROW
WHEN TRA_NOMBROKER= 'DUPONT'
BEGIN
IF :NEW.TRA_MONTANT > 10000000 THEN
            :NEW.TRA_MONTANT := 10000000;
ENDIF;
END;
```

5.8 Restrictions sur les triggers

Les opérations définies au sein du corps d'un trigger "Row Level" doivent respecter différentes restrictions que nous allons examiner :

- Tout trigger "Row Level " d'insertion, de mise à jour ou de suppression, ne peut en aucun cas effectuer, au cours de son traitement, **une opération de lecture** sur la table définie

[9] Personne effectuant des transactions boursières pour le compte de particuliers ou d'entreprises.

comme cible de l'instruction SQL associée. Le trigger suivant, dont le but est de placer, dans la variable *compteur* le nombre de clients après chaque opération d'insertion, de suppression ou de mise à jour est invalide puisqu'il effectue une opération de lecture sur la table concernée.

```
CREATE OR REPLACE TRIGGER CLIENTS_AIUDR[10]
AFTER INSERT OR UPDATE OR DELETE ON CLIENTS
FOR EACH ROW
Declare compteur number;[11]
BEGIN
Select count(*) into compteur from CLIENTS;
END;
```

- Tout trigger de mise à jour ou d'insertion défini sur une table "enfant" ne peut mettre à jour ou insérer, soit une colonne d'une table "parent", soit un enregistrement de la table "parent" qui est référencé par une colonne de la table "enfant" ou de la table étant modifiée par l'instruction d'insertion ou de mise à jour. Supposons que TRA_CLI_NUMID défini comme clé étrangère faisant référence à la clé primaire CLI_NUMID de la table clients. Le trigger ci-dessous défini sur la table transactions est invalide puisqu'il modifie une clé "parent" durant une instruction d'insertion ou de mise à jour sur la table "enfant".

```
CREATE OR REPLACE TRIGGER TRANSACTIONS_AIR
AFTER INSERT OR UPDATE ON TRANSACTIONS
FOR EACH ROW
BEGIN
        UPDATE CLIENTS set CLI_NUMID= TRA_CLI_NUMID where TRA_CLI_NUMID =1 ;
END;
```

Ce même trigger pourrait validement lire toutes les colonnes de la table *clients* et pourrait également mettre à jour toutes les autres colonnes de la table *transactions* à l'exception de la colonne de référence TRA_CLI_NUMID.

- Tout trigger de mise à jour ou de suppression associé à une table "parent" ne peut modifier un enregistrement ou une colonne d'une table "enfant" qui référence une colonne ou une colonne de la table "parent" associée à la déclaration du trigger. Si nous reprenons l'exemple précité, un trigger qui tenterait de modifier une colonne de la table *transactions* durant une opération UPDATE ou DELETE sur la table *clients* est invalide; même si la clé étrangère est déclarée avec l'option DELETE CASCADE[12].

```
CREATE OR REPLACE TRIGGER CLIENTS_AIR
AFTER INSERT OR DELETE ON CLIENTS
FOR EACH ROW
BEGIN
        UPDATE TRANSACTIONS set TRA_CLI_NUMID = CLI_NUMID where CLI_NUMID=1;
END;
```

- Tout trigger défini sur un événement de suppression ne peut lire un enregistrement d'une table qui est en cours de modification conformément à une règle "DELETE CASCADE".

[10] Convention d'écriture d'un trigger "After" de type "Row Level" se déclenchant pour des opérations d'insertion, de suppression ou de mise à jour.
[11] Fonction permettant de définir une variable compteur de type NUMBER.
[12] Cette notion est définie en annexe P.115.

Ceci implique qu'un trigger row-level associé à la table "parent" ne peut effectuer aucune opération de lecture sur une table "enfant" durant l'événement associé au trigger "parent". De même, aucun trigger relatif associé à une table "enfant", déclenché suite à l'exécution de la suppression en cascade, ne peut effectuer des commandes de lecture sur elle-même ou sur d'autres colonnes "enfant" durant l'opération de suppression sur la table "parent". En effet, la violation de cette règle entraîne une erreur du type "Table is mutating"[13].

5.9 Les triggers "Instead of"

Les triggers "instead of" fournissent une manière transparente de transformer les vues qui ne peuvent pas être modifiées directement par des instructions DML (INSERT, DELETE, UPDATE). Ces triggers sont appelés "instead of" parce que, contrairement aux autres types de triggers, Oracle déclenche le trigger au lieu d'exécuter l'opération de déclenchement. Le trigger effectue des opérations de mise à jour, de suppression ou d'insertion directement sur les données sous-jacentes. Il est possible d'écrire des opérations d'UPDATE, d'INSERT et de DELETE pour chaque vue et les triggers travaillent en arrière-plan pour que les actions s'opèrent correctement. Par défaut, les triggers "instead of" sont activés pour chaque ligne.

Les modifications de vues sont un réel problème pour les SGBD car, comme nous l'avons spécifié au chapitre 2, toute modification de vue (suppression, création et modification) engendre des changements dans la base et les conséquences de ces changements peuvent entraîner une certaine ambiguïté. Pour pallier à ces problèmes, certaines restrictions doivent être imposées pour qu'une vue puisse être modifiable[14]. Les triggers "instead of" peuvent être employés pour la gestion des vues qui ne sont pas modifiables autrement. Une vue est déclarée modifiable s'il est possible d'effectuer sur la vue des opérations de mise à jour, de suppression ou de création sans utiliser les triggers "instead of" et qu'elle est conforme aux restrictions émises. La vue ne peut pas posséder dans sa spécification d'opérateurs d'ensemble, de fonctions de regroupement, de clauses "GROUP BY, CONNECT BY ou START WITH, d'opérateur DISTINCT et enfin, la vue ne doit pas posséder de jointure.

5.10 Exécution des triggers

Pour les triggers activés (statut ENABLE), Oracle réalise certaines opérations :

- Il exécute les triggers de chaque type selon une séquence planifiée de déclenchement, lorsque au moins deux triggers sont déclenchés pour une opération donnée.
- Il effectue une vérification des contraintes d'intégrité à plusieurs moments en fonction du type de triggers et garantit que les triggers ne compromettent pas les contraintes d'intégrité déclarées.
- Il fournit des vues cohérentes (en lecture) pour des requêtes et des contraintes.
- Il assure la gestion des dépendances entre les triggers et les objets référencés au sein de l'action du trigger.

[13] Le concept de "table mutante" est défini p.78.
[14] Les conditions permettant de déclarer une vue modifiable, sont exposées p.26.

- Dans le cas de bases distribuées, il utilise le mécanisme du "two phase commit[15]" si un trigger met à jour une table distante.
- Il déclenche, selon un ordre indéfini, tous les triggers déclarés pour un même type d'événements.

Le modèle d'exécution des triggers et la vérification des contraintes d'intégrité peuvent se construire comme suit :

Supposons une simple opération de mise à jour (UPDATE). Celle-ci peut déclencher quatre types de triggers : BUS (Before Update Statement), BUR (Before Update Row), AUS (After Update Statement), AUR (After Update Row). Lorsqu'un trigger se déclenche, les vérifications des contraintes d'intégrité doivent être opérées. En outre, l'action des triggers peut déclencher d'autres triggers (triggers en cascade).

Oracle utilise un modèle d'exécution bien précis pour assurer la séquence des déclenchements successifs des triggers et la vérification des contraintes d'intégrité en parallèle. Il se décrit comme suit pour une opération d'UPDATE :

1. Exécution de tous les triggers BUS qui s'appliquent à la mise à jour.
2. Itérer pour chaque ligne modifiée par l'opération de mise à jour
   a) Exécuter tous les triggers BUR qui s'appliquent à la mise à jour
   b) Verrouiller et effectuer les changements de l'enregistrement concerné et effectuer une vérification des contraintes d'intégrité. Le verrou sera délié lorsque la transaction sera terminée (commit).
   c) Exécuter tous les triggers AUR qui s'appliquent à la mise à jour
3. Terminer la vérification des contraintes d'intégrité qui aurait été différée.
4. Exécution de tous les triggers AUS qui s'appliquent à la mise à jour.

La définition de ce modèle d'exécution est récursif. En effet, l'exécution de l'action d'un trigger peut déclencher d'autres triggers.

Ce modèle suppose que les actions et les vérifications se sont bien déroulées. En revanche, si une exception est soulevée au cours du traitement, toutes les actions effectuées (par l'opération elle-même mais également par celle des triggers) et associées à l'événement déclencheur doivent être annulées afin de revenir à la situation initiale. De cette manière, les contraintes d'intégrité ne peuvent être compromises par les triggers. Le modèle d'exécution tient donc compte de l'ensemble des contraintes d'intégrité et rejette les triggers qui violent les contraintes déclaratives d'intégrité.

5.11 Les Triggers versus contraintes déclaratives[16]

Plusieurs courants ont étudié les différentes techniques d'intégration des contraintes dans les bases de données. Si certains présentent le modèle ECA comme unique mécanisme pour

[15] Ce mécanisme permet d'approuver une transaction dans sa globalité, lorsque chaque base a effectué son approbation. En cas d'annulation provenant d'un base, la transaction est entièrement annulée.

[16] Nous présentons, en annexe, les cinq types de contraintes déclaratives proposées par Oracle.

intégrer les contraintes, les permissions et certaines logiques au sein d'une base de données, d'autres suggèrent que les contraintes déclaratives suffisent pour appliquer ces mêmes contraintes. En effet, Il y a plusieurs raisons techniques et pratiques qui justifient l'utilisation des contraintes déclaratives en lieu et place de triggers.

En réalité, le modèle de bases de données actives qui est le plus utilisé, doit intégrer conjointement les triggers procéduraux (modèle ECA) et le modèle des contraintes déclaratives tel qu'il est défini par le standard SQL international.

Les contraintes déclaratives, seules, ne peuvent suffire à supporter toutes les exigences d'une base de données active. Il est plus couramment admis que les triggers puissent être utilisés pour mettre en pratique des contraintes algorithmiques relativement complexes sur certaines colonnes lorsque les contraintes déclaratives de type CHECK deviennent trop difficiles à définir. En outre, les contraintes déclaratives ne peuvent effectuer certaines vérifications de sécurité, de contrôle sur les connexions, sur les utilisateurs. La meilleure manière de comparer est d'analyser la puissance des triggers comme substitut aux contraintes procédurales.

Les syntaxes des contraintes procédurales de type CHECK, par exemple, permettent d'associer, à une ou plusieurs colonnes, une fonction exécutoire devant être respectée lors de chaque instruction d'insertion ou de mise à jour. Un exemple relativement simple est celui d'une contrainte qui porte sur la colonne *CompteCourant* d'une table *COMPTES*. Cette contrainte exige que le compte courant soit crédité d'au moins 1000 francs.
L'instruction ci-dessous présente l'écriture de cette contrainte sous la forme déclarative :

```
CREATE TABLE COMPTES ( numcompte NUMBER(12) PRIMARY KEY,
    ...
    comptecourant NUMBER (10,2) CHECK (comptecourant >1000),
    ...) ;
```

Cependant, les contraintes de type CHECK, malgré leur efficacité, ne sont pas assez puissantes pour supporter des contraintes trop complexes. Ces contraintes complexes peuvent être la vérification associée à des données situées dans différentes tables et reliées par diverses associations. Par exemple, comment intégrer la règle qui octroie un crédit de caisse de 25.000 francs seulement si la somme de tous les comptes du clients (compte courant, épargne..) est positive? On remarque assez rapidement les limites des contraintes CHECK. Cependant, modéliser l'ensemble des contraintes déclaratives par des triggers peut alourdir inutilement la charge de la base par les déclenchements successifs de triggers (en cas de déclaration de nombreux triggers) et nuire à la performance de la base.

En outre, la modélisation des contraintes déclaratives est relativement simple à comprendre (ce qui n'est pas toujours le cas des triggers) et a été suffisamment étudiée pour qu'elle devienne un standard dans la déclaration de contraintes. Il est plus opportun de considérer les triggers comme un complément des contraintes déclaratives. Utiliser les triggers uniquement dans les cas où les contraintes déclaratives ne peuvent s'appliquer est, en quelque sorte, le maître mot que préconisent la plupart des systèmes actuels.

Il est possible de combiner les triggers et les contraintes déclaratives pour définir et renforcer les règles générales d'intégrité. Oracle recommande d'utiliser les triggers de base de données pour appliquer des contraintes sur les données dans les situations suivantes :

- Lorsque les règles de contraintes d'intégrité requises ne peuvent être appliquées en utilisant les contraintes d'intégrité suivante :

1.	La clé non nulle et unique	NOT NULL, UNIQUE KEY
2.	La clé primaire	PRIMARY KEY
3.	La clé étrangère	FOREIGN KEY
4.	Vérification via la commande CHECK	CHECK
5.	Mise à jour en cascade	UPDATE CASCADE
6.	Remplacement par la valeur NULL	UPDATE AND DELETE SET NULL
7.	Remplacement par une valeur par défaut	UPDATE AND DELETE SET DEFAULT

- Pour imposer l'intégrité référentielle lorsque des tables "enfants et parents" se trouvent sur des nœuds différents dans le cadre de bases de données distribuées.
- Pour imposer des règles complexes non définissables par des contraintes d'intégrité.

Certains auteurs ont tendance à affirmer que, pour des raisons de gestion, les triggers doivent avoir priorité sur les contraintes déclaratives. Si du point de vue de la gestion des règles, les triggers sont plus flexibles, il faut garder à l'esprit que ceux-ci sont plus difficiles à implémenter. Nous optons pour une solution où les règles (stables) doivent être implémentées par des contraintes déclaratives et les triggers prennent le relais des contraintes déclaratives. Une règle pouvant être modélisée par une contrainte déclarative mais devant subir régulièrement des adaptations, sera de préférence modélisée sous forme de trigger(s).

5.12 Limitations des triggers

Nous pouvons brièvement parler des limitations[17] des triggers. Ces limitations constituent actuellement les principaux pôles de recherche. Leurs objectifs est de développer des techniques encore plus puissantes permettant de repousser ces limites.

➢ Le langage PL/SQL associé au trigger possède un "caractère expressif limité" surtout dans la partie spécification d'événement. En effet, les règles que l'on veut spécifier doivent parfois être découpées en plusieurs triggers pour arriver à leurs fins. Ce découpage peut alourdir le système et entraîner des déclenchements en série.

➢ Le modèle d'exécution des triggers peut être trop limité pour certaines applications. Nous avons vu précédemment que les restrictions des triggers "row-level" imposent des limitations relativement stricts. De même, les contrôles de transactions (commit, rollback et savepoint)sont interdits au sein des triggers. Par manque de pratique et de connaissance des triggers, les développeurs préfèrent adopter une solution "ad-hoc". Soulignons quelques questions qui reviennent régulièrement : Quels sont les critères pour décider de choisir une modélisation par des procédures stockées ou par triggers? Quelles sont les conditions nécessaires pour vérifier que les triggers sont corrects et que leurs déclenchements se termineront dans tous les cas? Comment déclarer des contraintes qui

[17] Kulkarni K. Mattos N., Cochrane R., "*Active Datase features in SQL3*", Monographs in Computer Sciences, springer, 1998.

exigent différents traitements en fonction de critères particuliers? Ces questions ne trouvent pas toujours des réponses ou du moins, si elles existent, elles sont peu adaptées aux exigences des utilisateurs.

- Il est difficile de valider un grand nombre de règles. Imaginons une série importante de triggers dont le but est de rectifier des erreurs d'encodage. Il est très difficile, malgré les produits existants, d'éviter certains déclenchements cycliques. Des techniques permettent de stopper ce phénomène. Mais comment l'éviter de manière anticipative?

- Il est difficile pour les utilisateurs de comprendre certains comportements de la base en présence de triggers. Des utilisateurs peuvent être très surpris face à certains comportements (correction automatique) modélisés par des triggers et insérés dans la base. De même, en cas d'erreurs, les messages à l'écran ne sont pas toujours très explicites. La lisibilité et la clarté sont des éléments à ne pas négliger.

- Les triggers doivent être plus sécurisés face aux accès non autorisés. Les systèmes actuels associent des privilèges aux utilisateurs pour pouvoir modifier les triggers. Ceci peut être problématique car le programmeur du trigger doit savoir quelles règles peuvent être déclenchées par les transactions qu'il écrit et plus particulièrement les actions qui peuvent être effectuées par ces triggers. Maintenant, les programmeurs doivent seulement avoir les privilèges d'exécution et non nécessairement ceux de lecture. En outre, en cas d'erreurs, une certaine confidentialité peut être présente face aux actions associées aux triggers de réparation.

- La séparation des transactions et des triggers peut perturber l'optimisation globale de l'application. Les phases de spécification des transactions et des triggers sont complètement séparées. Cette situation peut suggérer "le problème de l'iceberg" puisque, d'une part, si l'on regarde dans la base de données, la partie visible de l'application est le schéma de la base de données incluant les triggers et la partie non visible, les transactions, et d'autre part, la situation est inverse pour le programmeur de l'application.

- Il est important d'apporter de plus amples informations sur les effets des triggers (effets sur les données, les relations associées..) et sur les moments de déclenchements. De cette manière, des statistiques sur les exécutions opérées, les accès réalisés et sur les données affectées pourraient apporter une aide appréciable sur la maintenance des règles. Des outils tentent également de fournir des informations de traçabilité des triggers.

- ...

Comme on peut le voir, les problèmes associés aux triggers sont encore importants mais les recherches et les études actuelles ont pour but de répondre à ces défaillances en proposant de nouvelles sémantiques, de nouvelles techniques de modélisation et des outils de développement et de diagnostic aux développeurs mais également aux utilisateurs. Face à cette analyse et malgré les critiques émises, il est relativement certain que les règles actives sont et seront assurément un produit d'avenir dans les systèmes de gestion de base de données.

La suite de notre analyse va tenter d'apporter une méthodologie de développement de triggers afin de mieux orienter les programmeurs dans leur démarche de modélisation.

Chapitre 6:

Les triggers et les procédures stockées: analyse et implémentation

6.1 Implémentation des contraintes d'intégrité

6.2 Transformation d'une base de données en "base d'objets"

6.3 Méthodologie de développement des triggers

6.3.1 Étapes du processus de développement

6.3.2 Identification des règles à implémenter de façon procédurale

6.3.3 Construire une liste des violations de contraintes ou une liste de vérification de contraintes

6.3.4 Donner la description fonctionnelle du trigger

6.3.5 Définir les erreurs d'erreur et les actions associées

6.3.6 Encapsuler les fonctionnalités dans un package "Constraints Package"

6.3.7 Analyser les conflits éventuels avec les contraintes déclaratives

6.3.8 Écrire les triggers et les tester

6.4 Algorithme du processus de développement d'un trigger

6.5 Les triggers de complexité non nulle

6.5.1 Les tables mutantes

6.5.2 Résolution du conflit

6.6 Utilisation des triggers et des procédures stockées

Chapitre 6 :

Les triggers et les procédures stockées : analyse et implémentation

6.1 Implémentation des contraintes d'intégrité

L'implémentation des règles stratégiques de gestion, comme nous l'avons déclaré au chapitre précédent, ne peut se faire complètement par les contraintes déclaratives. En effet, prenons ces trois contraintes à titre d'exemples :

- Les taux de change ne peuvent varier de plus de 5%.
- La somme de toutes les transactions (actions, obligations..) en cours doit être inférieure à un montant fixé.
- Seul le directeur financier peut effectuer des calculs de consolidation; uniquement entre le 25 et la fin de chaque mois.

Il existe trois différentes manières pour implémenter ces règles, chacune d'elles constituant une solution de type procédural.

1. Via un programme d'application : on peut imaginer, pour la modélisation de la deuxième contrainte, un programme C qui exécute une transaction qui débute par une insertion ou une modification d'une transaction (INSERT ou UPDATE dans la table TRANSACTIONS), récupère, ensuite, la somme des transactions en cours, et compare le résultat à la limite fixée. La violation de la règle implique le ROLLBACK de la transaction et l'envoi d'un message d'erreurs.

2. Via un trigger

```
CREATE OR REPLACE TRIGGER TRANSACTIONS_AUIS
AFTER  INSERT OR UPDATE OF MONTANT ON TRANSACTIONS
TOTAL NUMBER;
LIM_MAX CONSTANT NUMBER = 100.000.000;
BEGIN
    SELECT SUM(MONTANT) INTO TOTAL FROM TRANSACTIONS;
    IF TOTAL > LIM_MAX THEN
    RAISE_APPLICATION_ERROR (-20100, 'Erreur Transactions');
    ENDIF;
END;
```

3. Via une procédure stockée : Cette même contrainte peut être exprimée sous la forme d'une procédure stockée qui serait appelée par les programmes applicatifs et qui, avant d'effectuer une mise à jour ou une insertion sur la table TRANSACTIONS, effectuerait les vérifications nécessaires. Dans ce cas précis, les applications ne devraient disposer d'aucun droit d'accès direct sur la table afin de les obliger à exécuter la procédure stockée.

Nous pouvons évaluer ces trois solutions en comparant la solution du programme d'application (premier cas) par rapport à celles du traitement intégré dans la base de données (trigger ou procédure stockée).

- Une contrainte ne sera vérifiée par un programme extérieur à la base de données que si tous les programmes applicatifs passent par lui pour effectuer le contrôle d'intégrité. Une contrainte vérifiée au niveau des triggers ou des procédures stockées, sera vérifiée automatiquement par tous les programmes.

- Lorsqu'il est nécessaire de déplacer une base de données vers un nouveau site, si les contraintes sont implémentées par des applications, celles-ci devront être déplacées également, voire réécrites complètement. Cette situation n'était sûrement pas désirée initialement. Ce transfert est techniquement plus complexe que l'opération de transfert de la base de données, elle-même, et des triggers et procédures stockées associés. Aucun changement n'est nécessaire pour le transfert des triggers et des procédures stockées.

- Le langage SQL est un langage standardisé : il permet une certaine indépendance vis-à-vis des gestionnaires de base de données. Les extensions procédurales à SQL sont, dès lors, propres à chaque SGDB. Les utiliser implique une perte de cette indépendance. Écrire des contraintes avec des langages standardisés comme Pascal ou C, dans ce contexte, reste peut-être une solution plus stable. Il est vrai que des triggers PL/SQL ne peuvent être intégrés à d'autres SGBD tels que SQL Server qui utilise, quant à lui, le langage TRANSACT-SQL.

Malgré cette dernière critique, les triggers et les procédures stockées restent un outil stratégique très performant. N'oublions pas que les entreprises, en général, appliquent une politique de système de bases de données uniforme. Les transferts de base de données s'opèrent dès lors dans des architectures relativement compatibles.

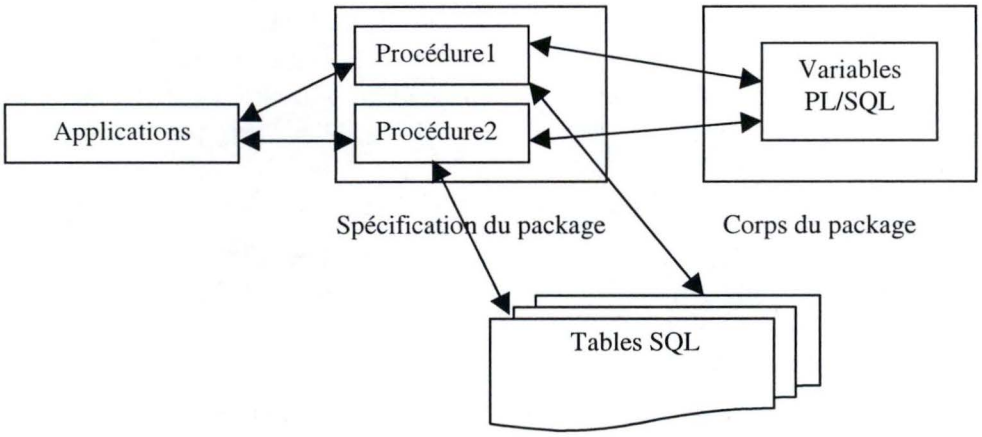
6.2 Transformation d'une base de données en "base d'objets"

L'ajout de mécanismes tels que les triggers et les procédures stockées au sein de la base de données, est susceptible de transformer celle-ci en "base d'objets". Cette vision est quelque peu osée mais l'écriture des PACKAGES en PL/SQL permet tout au plus de simuler une certaine encapsulation des données et des "méthodes" (procédures et fonctions définies au sein du package). Le langage PL/SQL, tout comme le C ou le PASCAL, ne fournit aucun mécanisme explicite d'encapsulation, d'héritage ou de polymorphisme.

Il est possible de simuler l'encapsulation des données et des méthodes puisque PL/SQL permet de définir des structures de données (SQL mais également PL/SQL). Les méthodes qui s'appliquent sur ces données sont définies par les procédures et les fonctions définies par les packages SQL. Il est dès lors possible de regrouper, dans un package PL/SQL, des procédures et des fonctions qui ont trait à une même structure de données.

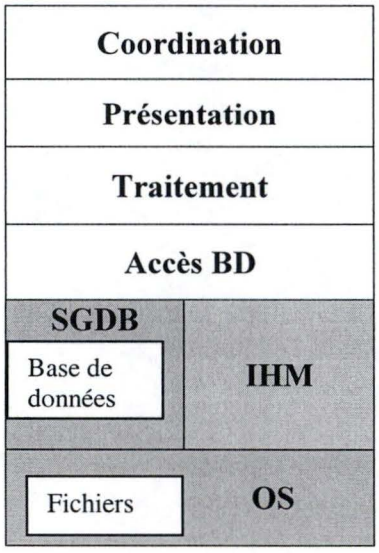
Cette structure peut être implémentée par une table SQL, par une variable PL/SQL ou, encore, par un combinaison de tables SQL et de variables PL/SQL. Il faut également obliger les programmes d'application à manipuler ces données uniquement au travers des procédures et des fonctions. En effet, il faut donc interdire tous les droits d'accès directs aux tables SQL, en leur refusant l'utilisation des commandes DML sur les tables concernées et en déclarant les

structures de données PL/SQL dans le corps du package et non dans la partie spécification. Le schéma résume cette technique de modélisation.



Un objet est défini aux yeux des programmes d'application. Si le processus exposé ci-dessus est réitéré jusqu'à ce que plus aucune table SQL ou variable PL/SQL ne soit directement accessible par ces derniers, alors une base d'objets est définie et est visible depuis ces programmes.

Avec la technique exposée, le module d'un programme d'application qui est du niveau le plus bas, c'est-à-dire qui réalise les accès à la base de données, a une vue plus abstraite de la base de données. Il n'a aucune connaissance des détails de l'implémentation des données sous la forme de tables SQL ou de variables PL/SQL. Il ne les voit que par l'interface qui lui est fournie par les procédures et les fonctions. Ceci suppose que l'architecture du programme ait été conçue selon le modèle classique.



Architecture du programme

Cette vue "objet" peut tout aussi bien être implémentée une couche plus haut en écrivant en C, Pascal, l'équivalent des packages SQL. Mieux, on peut, à ce niveau, utiliser n'importe quel langage véritablement orienté objet tel que C++ ou JAVA, et donc définir aux yeux du module de niveau supérieur (le module traitement de la figure), une base d'objets nettement plus adaptée dans la philosophie orienté objet que celle que l'on peut définir avec les packages SQL.
Si la définition d'une base "objets" peut être faite pour le module d'accès à la base de données, il est souhaitable que les couches supérieures soient du même type. Ce choix relève d'un choix d'architecture concernant l'application entière.

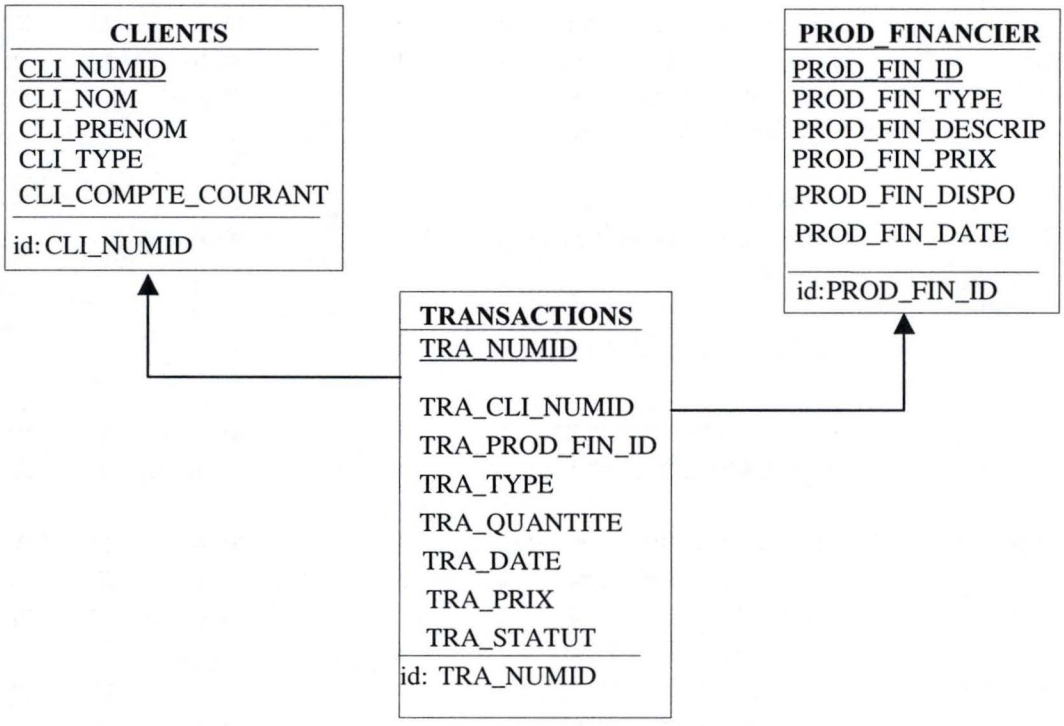
6.3 Méthodologie de développement des triggers

Une approche systématique basée sur des méthodes standards est essentielle pour le développement des triggers. En effet, ce développement peut poser certains problèmes. Une analyse de la cohérence et de l'intégration des procédures stockées dans l'ensemble des contraintes émises doit être opérée afin d'éviter l'apparition de conflits lors de l'exécution de triggers. La maintenance et la flexibilité du code procédural impliquant une approche modulaire dans le développement des procédures et des fonctions doivent être intégrées dans la création des triggers. Une méthode de conception de triggers s'impose donc car si les utilisateurs possèdent certaines informations sur les processus ou les programmes en exécution sur leur partie, ils ont, par contre, très peu de connaissances sur les opérations effectuées par les triggers.

Ces opérations sont généralement exécutées de manière sous-jacente. Pour les bases de données relativement grandes, devant intégrer des règles stratégiques relativement complexes, il est important de disposer de méthodes de développement de triggers pour assurer leur création, leur implémentation, et leur maintenance.

Ce chapitre tente de répondre à ces objectifs. Les références de cette méthodologie de développement sont les chapitres 15 à 17 du livre Owens, Kevin T.,"*Building Intelligent Databases with Oracle PL/SQL, Triggers, & Stored Procedures*", Prentice Hall, 1998.

Pour ce faire, nous travaillerons sur une application simplifiée de gestion de portefeuille décrite ci-dessous.



6.3.1 Étapes du processus de développement

Le processus de développement des triggers peut se décomposer en sept étapes :

1. Identification des règles à implémenter de façon procédurale;
2. Construire une liste des violations de contraintes ou une liste de vérification de contraintes;
3. Donner la description fonctionnelle du trigger;
4. Définir les erreurs et les actions associées;
5. Encapsuler les fonctionnalités dans un package appelé le "Constraints Package";
6. Analyser les conflits éventuels avec les contraintes déclaratives;
7. Écrire les triggers et les tester.

Nous allons décrire chaque étape du processus de développement à travers notre exemple simplifié.

6.3.2 Identification des règles à implémenter de façon procédurale.

La première étape à accomplir est d'identifier les différentes contraintes à insérer et de trier ces règles afin d'éviter d'utiliser des procédures pour implémenter ces règles alors que des contraintes déclaratives peuvent effectuer cette opération d'une manière beaucoup plus simple. Pour cette opération, il est indispensable de bien connaître les capacités de modélisation des contraintes déclaratives (CHECK, PRIMARY KEY, FOREIGN KEY..), mais également le modèle de données et les méthodes de design. En effet, une bonne compréhension du langage PL/SQL, ainsi que des outils existants permettant de développer des triggers PL/SQL, est un atout non négligeable. Une mauvaise compréhension ou/et une mauvaise analyse peut mener à la modélisation de multiples triggers associés à une seule règle ou à des triggers très complexes.

Nous voulons insérer ces différentes règles sur notre application de gestion de portefeuille :

Règle 1: On ne peut modifier la date d'une transaction (TRA_DATE).
Règle 2: Les produits financiers sont de type (action, obligation, swap, sicav,..call et autres). Cette liste est limitée à ces composants.
Règle 3: Un numéro d'un client est unique et obligatoire (CLI_NUMID).
Règle 4: Les domaines de TRA_STATUT et de TRA_TYPE sont respectivement ("en cours" ou "cloturee") et ("achat ou "vente").
Règle 5: Une transaction de statut (TRA_STATUT) "en cours" ne peut être supprimée.
Règle 6: Un produit financier de type "action" ou "obligation" doit obligatoirement avoir une description (PROD_FIN_DESCRIP).

On peut voir que les règles 2 et 4 peuvent être implémentées par des contraintes de type CHECK (PROD_FIN_TYPE VARCHAR(20) **CONSTRAINT CHECK (PRO_FIN_TYPE IN ('action', 'obligation', 'swap', .., 'call','autres'))**,. La règle 3 correspond à une contrainte de clé primaire (CLI_NUMID NUMBER(10) **CONSTRAINT PK_CLIENTS PRIMARY KEY ,).**

Notons qu'il serait préférable de modéliser la règle par un trigger si la liste des produits financiers devait subir régulièrement des modifications.

Les règles 1, 5 et 6 seront implémentées de façon procédurale et feront l'objet de notre analyse.

6.3.3 Construire une liste des violations de contraintes (Constraints Violation List CVL) ou une liste de vérification de contraintes (Constraints Enforcement List CEL)

Cette deuxième étape a pour objectif de construire une liste des violations de contraintes (ou une liste de vérification de contraintes). Elle recense l'ensemble des événements, des actions et des commandes qui peuvent violer les règles que l'on tente d'implémenter. On effectue à ce niveau un lien entre chaque règle identifiée et le(s) trigger(s) qui va(ont) l'implémenter. Pour cette raison, chaque violation ne peut associer plus d'une table et pas plus d'une opération (INSERT, DELETE et UPDATE).

Pour effectuer cette étape, il est préférable de relever, pour chaque règle, tous les événements et éliminer tous ceux qui sont sans menace pour l'intégrité de la base.

La liste de violation de contraintes s'établit comme suit :

- UPDATE sur la table pour changer l'attribut TRA_DATE. (violation de la règle 1).
- DELETE sur la table TRANSACTIONS, qui a l'attribut TRA_STATUT "en cours" (violation de la règle 5).
- INSERT sur la table PROD_FINANCIER qui a un attribut PROD_FIN_TYPE "action" ou "obligation", et qui ne spécifie pas l'attribut PROD_FIN_DESCRIP (violation de la règle 6).
- UPDATE sur la table PROD_FINANCIER qui a un attribut PROD_FIN_TYPE "action" ou "obligation", et qui met à jour PROD_FIN_DESCRIP à NULL (violation de la règle 6).
- UPDATE sur la table PROD_FINANCIER qui a un attribut PROD_FIN_DESCRIP à NULL, et qui met à jour la valeur de PROD_FIN_TYPE par "action" ou "obligation" (violation de la règle 6).

A partir de cette analyse, on peut dégager les quatre événements SQL (tables + commandes DML) qui sont à prendre en considération :

UPDATE TRANSACTIONS	DELETE TRANSACTIONS
INSERT PROD_FINANCIER	UPDATE PROD_FINANCIER

6.3.4 Donner la description fonctionnelle du trigger

Pour chaque événement, cette étape exige d'effectuer une description fonctionnelle du trigger (Trigger Functional Description - TDF). En effet, pour chaque trigger, il est indispensable de spécifier les données nécessaires pour effectuer le traitement (valeur de corrélation, données de table) et le type du trigger (Row ou Statement Level). La description fonctionnelle rassemble la condition de déclenchement (via une clause WHEN ou une condition insérée dans le corps de l'action) du trigger et les données associées au traitement.

Examinons la description fonctionnelle de chaque événement identifié au point précédent :

- UPDATE TRANSACTIONS

  Description : ce trigger a pour objectif de détecter une violation si les valeurs de corrélation :OLD.TRA_DATE et :NEW.TRA_DATE sont différentes.

  Données nécessaires : :OLD.TRA_DATE et :NEW.TRA_DATE

  Type du Trigger : Row Level puisque les valeurs de corrélation sont utilisées.

- DELETE TRANSACTIONS

  Description : ce trigger a pour objectif de détecter une violation si l'enregistrement que l'on veut effacer, a un attribut TRA_STATUT de valeur "en cours".

  Données nécessaires : :OLD.TRA_STATUT

  Type du Trigger : Row Level

- INSERT PROD_FINANCIER

  Description : ce trigger a pour objectif de détecter une violation si un nouveau produit financier inséré a un NEW.PROD_FIN_TYPE de valeur "action" ou "obligation" avec :NEW.PROD_FIN_DESCRIP de valeur NULL.

  Données nécessaires : :NEW.PROD_FIN_DESCRIP et :NEW.PROD_FIN_TYPE

  Type du Trigger: Row Level

- UPDATE PROD_FINANCIER

  Description : ce trigger a pour objectif de détecter une violation si un produit financier modifié a un NEW.PROD_FIN_TYPE de valeur "action" ou "obligation" avec :NEW.PROD_FIN_DESCRIP de valeur NULL.

  Données nécessaires : :NEW.PROD_FIN_DESCRIP et :NEW.PROD_FIN_TYPE

  Type du Trigger : Row Level

Il est possible, lors de cette analyse, de trouver des redondances de fonctionnalités (par exemple, les deux derniers triggers ont la même description). Il en résultera deux triggers qui appelleront la même procédure stockée implémentant la description fonctionnelle. A partir de cette analyse, il faut toujours avoir à l'esprit si l'interprétation déduite est toujours cohérente par rapport aux intentions des décideurs. En outre, cette analyse peut suggérer des changements dans les tables de données (additions de tables ou de colonnes) mais également la réutilisation de certaines procédures stockées déjà écrites.

6.3.5 Définir les erreurs et les actions associées

Cette étape a pour objectif de définir les messages d'erreurs pour chacune des violations constatées. On écrit un package à cet effet où les messages d'erreurs seront définis en réponse aux violations constatées par les triggers. Conjointement, pour les triggers qui n'affichent pas uniquement des messages d'erreurs, les actions à exécuter sont définies. Nous allons présenter un exemple de ce package recensant les messages d'erreurs pour les triggers que nous analysons actuellement.

```
PACKAGE ERREURS1 IS
    UP_TRA_DATE CONSTANT INTEGER := -20001;
    DEL_TRA_VIOLATION CONSTANT INTEGER := -20002;
        UP_INS_PROD_FIN_DESCRIP_MANQ CONSTANT INTEGER := -20003;
```

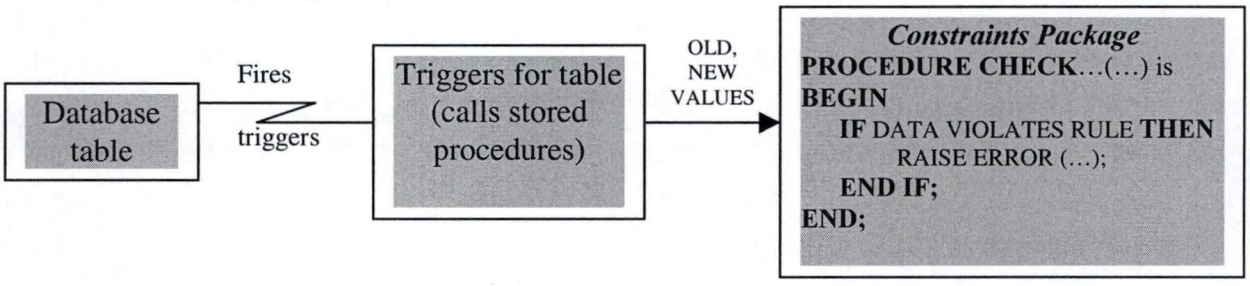
[1] Ce package n'a pas besoin d'un corps.

```
    UP_TRA_DATE_MSG VARCHAR2(55) NOT NULL := 'IMPOSSIBLE DE METTRE A JOUR
    LA DATE D'UNE TRANSACTION';
    DEL_TRA_VIOLATION_MSG VARCHAR2(66) NOT NULL :='IMPOSSIBLE DE
    SUPPRIMER UNE TRANSACTION AVEC UN STATUT EN COURS';
    UP_INS_PROD_FIN_DESCRIP_MANQ_MSG VARCHAR2(66) NOT NULL :='UN PRODUIT
    FINANCIER ACTION OU OBLIGATION DOIT AVOIR UNE DESCRIPTION';
END ERREURS;
```

Certains triggers implémentent une règle en modifiant les valeurs de corrélation pour les triggers "row-level" et en exécutant des commandes supplémentaires d'insertion, de mise à jour ou de suppression. Pour ces triggers, cette étape est utile pour préciser la logique PL/SQL à exécuter quelque soit l'événement déclenché.

6.3.6 Encapsuler les fonctionnalités dans un package appelé le "Constraints Package"

Cette étape crée la spécification et le corps du package des contraintes. Ce dernier est un package PL/SQL qui encapsule les règles définies de manière procédurale. Chaque procédure dans le package peut accepter des arguments qui sont passés comme valeurs de corrélation. La procédure détermine les occurrences des violations et soulève les exceptions associées. Les informations provenant de l'analyse fonctionnelle constitue la base de l'écriture de la spécification (interface) du package et le corps du package résulte de l'étude du point précédent. La figure ci-dessous présente l'association existante entre les triggers et les procédures dans le package des contraintes.



Package des contraintes encapsulant les fonctionnalités des triggers[2]

Le package des contraintes permet d'encapsuler différents algorithmes au sein de programmes PL/SQL. Il possède les caractéristiques suivantes :

- Il permet de cacher la complexité des algorithmes.
- Il contient des procédures qui peuvent soulever des exceptions, annuler et corriger les valeurs de corrélation et exécuter des commandes d'insertion, de mise à jour ou de suppression.
- Il peut exécuter des commandes DML.
- Les paramètres des procédures sont toujours du type *%TYPES (COLUMN%TYPE, %ROWTYPE)*.
- Plusieurs packages de contraintes peuvent être insérés dans le schéma.

Les avantages de cette modélisation en packages sont les suivants :

- Les procédures peuvent être testées dans un environnement SQL*Plus.

[2] Owens, Kevin T.,"*Building Intelligent Databases with Oracle PL/SQL, Triggers, & Stored Procedures*", Prentice Hall, p.433, 1998.

- Les modifications des règles peuvent être effectuées sans intervenir dans la modélisation des autres règles. La maintenance est dès lors simplifiée.
- Il est possible de réutiliser le code modulaire pour mettre en application des triggers associés à d'autres tables.
- Lorsque le nom du package est conforme à certaines conventions, les règles implémentées par les contraintes procédurales sont plus facilement gérables.

Pour notre première règle, on peut créer les packages de contraintes suivants pour les deux tables transactions et prod_financier.

```
/*****************************************************************************************
*                          TRANSACTIONS CONSTRAINTS PACKAGE SPEC. AND BODY              *
*****************************************************************************************/
CREATE OR REPLACE PACKAGE TRANSACTIONS_CONS_PKG IS
        PROCEDURE CHECK_TRA_DATE
                ( OLD_DATE : TRANSACTIONS.TRA_DATE%TYPE,
                  NEW_DATE : TRANSACTIONS.TRA_DATE%TYPE);
        PROCEDURE CHECK_DEL_TRANSAC
                ( TRANSACTION : TRANSACTIONS.TRA_STATUT%TYPE);
        END TRANSACTIONS_CONS_PKG;

CREATE OR REPLACE PACKAGE BODY TRANSACTIONS_CONS_PKG IS
        PROCEDURE CHECK_TRA_DATE
                ( OLD_DATE : TRANSACTIONS.TRA_DATE%TYPE,
                  NEW_DATE : TRANSACTIONS.TRA_DATE%TYPE);
        BEGIN
                  IF ( TRUNC(OLD_DATE) <> TRUNC (NEW_DATE) THEN
                    RAISE_APPLICATION_ERROR (ERREURS.UP_TRA_DATE,
                        ERREURS.UP_TRA_DATE_MSG);
                  ENDIF;
        END CHECK_TRA_DATE

        PROCEDURE CHECK_DEL_TRANSAC
        (       STATUT : TRANSACTIONS.TRA_STATUT%TYPE) IS
        BEGIN
                IF STATUT= 'en cours' THEN
                   RAISE_APPLICATION_ERROR (ERREURS.DEL_TRA_VIOLATION,
                       ERREURS.DEL_TRA_VIOLATION_MSG);
                ENDIF;
        END CHECK_DEL_TRANSAC;
END TRANSACTIONS_CONS_PKG;
/*****************************************************************************************
*                           PROD_FINANCIER CONSTRAINTS PACKAGE SPEC. AND BODY           *
*****************************************************************************************/
CREATE OR REPLACE PACKAGE PROD_FINANCIER_CONS_PKG IS
        PROCEDURE CHECK_PROD_FIN_DESCRIP
                ( DESCRIP : PROD_FIN.DESCRIP%TYPE,
                  TYPE : PROD_FIN.TYPE%TYPE);
END PROD_FINANCIER_CONS_PKG;

CREATE OR REPLACE PACKAGE BODY PROD_FINANCIER_CONS_PKG IS
        PROCEDURE CHECK_PROD_FIN_DESCRIP
                ( DESCRIP : PROD_FIN.DESCRIP%TYPE,
                  TYPE : PROD_FIN.TYPE%TYPE);
        BEGIN
                  IF ( DESCRIP IS NULL) AND ('action' = TYPE OR 'obligation' = TYPE) THEN
                    RAISE_APPLICATION_ERROR (ERREURS.UP_INS_PROD_FIN_DESCRIP_MANQ,
                        ERREURS.UP_INS_PROD_FIN_DESCRIP_MANQ_MSG);
                  ENDIF;
        END CHECK_PROD_FIN_DESCRIP;
END PROD_FINANCIER_CONS_PKG;
```

6.3.7 Analyser les conflits éventuels avec les contraintes déclaratives

Cette étape est nécessaire pour les triggers qui effectuent des opérations de lecture ou d'écriture sur d'autres tables que celle associée au trigger. Ce sujet sera étudié par la suite[3] car jusqu'à présent, nous avons considéré uniquement des triggers simples ou de complexité nulle.

6.3.8 Écrire les triggers et les tester

Ayant inséré les fonctionnalités des triggers dans un package, il reste encore à développer le code des triggers et de les tester.

```
/*****************************************************************************************
*                          TRANSACTIONS AFTER UPDATE / DELETE ROW TRIGGER                *
*****************************************************************************************/
        CREATE OR REPLACE TRIGGER TRANSACTIONS_AUR
        AFTER UPDATE ON TRANSACTIONS
        FOR EACH ROW
        BEGIN
                TRANSACTIONS_CONS_PKG.CHECK_TRA_DATE (:OLD. TRA_DATE , NEW. TRA_DATE);
        END;
        CREATE OR REPLACE TRIGGER TRANSACTIONS_ADR
        AFTER DELETE ON TRANSACTIONS
        FOR EACH ROW
        BEGIN
                TRANSACTIONS_CONS_PKG.CHECK_DEL_TRANSAC (:OLD. TRA_STATUT);
        END;
/*****************************************************************************************
*                      PROD_FINANCIER AFTER INSERT OR UPDATE: ROW TRIGGER                *
*****************************************************************************************/
        CREATE OR REPLACE TRIGGER PROD_FINANCIER_AIUR
        AFTER INSERT OR UPDATE ON PROD_FINANCIER
        FOR EACH ROW
        BEGIN
                PROD_FINANCIER_CONS_PKG. CHECK_PROD_FIN_DESCRIP (:NEW.PROD_FIN_DESCRIP,
                :NEW.PROD_FIN.TYPE);
        END;
```

L'écriture des triggers terminée, il faut encore effectuer tous les tests afin de vérifier l'implémentation de la règle par rapport aux exigences de la règle spécifiée.
Dans notre cas, plusieurs cas sont à tester :

Pour la règle 1 :

- Effectuer une insertion d'une transaction; cette opération doit réussir.
- Effectuer une suppression de transaction; cette opération doit réussir.
- Effectuer une mise à jour de la date d'une transaction en la remplaçant par la même date; cette opération doit réussir.
- Effectuer une mise à jour de la date d'une transaction en la remplaçant par une autre date; cette opération doit échouer et afficher un message d'erreur 'IMPOSSIBLE DE METTRE A JOUR LA DATE D'UNE TRANSACTION'.

  ```
  SQL> EXECUTE TRANSACTIONS_CONS_PKG.CHECK_TRA_DATE (SYSDATE, SYSDATE-1);
  ORA-20001: IMPOSSIBLE DE METTRE A JOUR LA DATE D'UNE TRANSACTION
  ```

- Effectuer les mêmes tests pour des insertions, des suppressions et des mises à jour associées sur plusieurs transactions.

Si les tests pour la règle 1 sont satisfaits, ils doivent également être réalisés pour les règles 5 et 6. Si des erreurs sont détectées, il faut effectuer les changements nécessaires et tester à nouveau le trigger modifié.

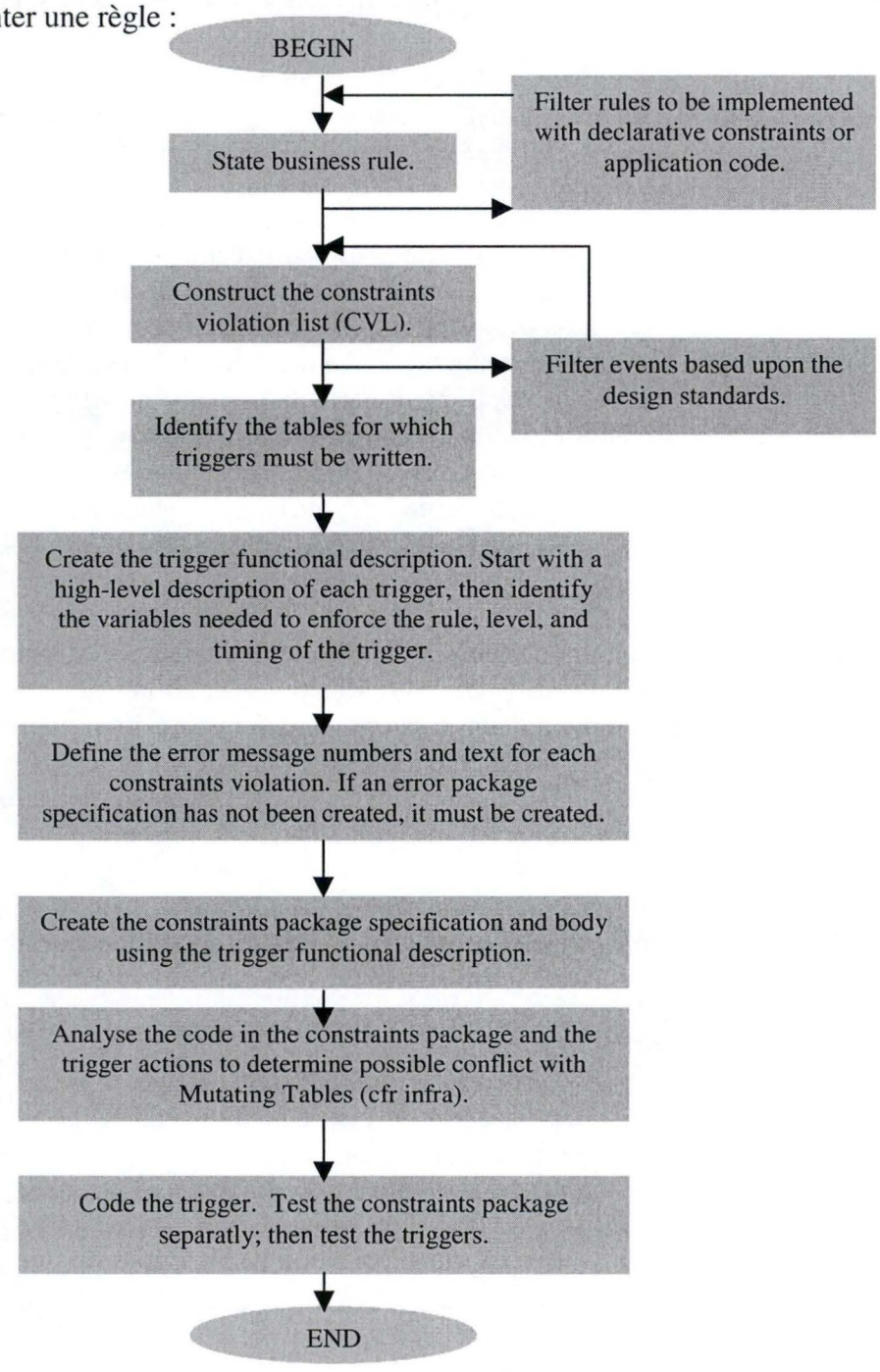
---

[3] Cfr P.78

Notons que la modélisation par des procédures stockées peut également se réaliser par des fonctions. Le choix entre les fonctions et les procédures stockées dépend des conditions de modélisation puisque la fonction doit obligatoirement renvoyer une valeur de retour.

6.4 Algorithme du processus de développement d'un trigger

Nous allons résumer la démarche que nous venons de présenter, en distinguant les étapes qui permettent d'implémenter une règle :



Etape du développement des triggers[4]

[4] Owens, Kevin T.,"*Building Intelligent Databases with Oracle PL/SQL, Triggers, & Stored Procedures*", Prentice Hall, p.438, 1998.

6.5 Les triggers de complexité non nulle

Jusqu'à présent, nous nous sommes concentrés sur les triggers simples ou de complexité nulle. En fonction de l'action qu'ils effectuent, les triggers peuvent être classés en trois catégories:

- *Les triggers simples* : Ces triggers ne contiennent aucune opération de SELECT, INSERT, DELETE ou UPDATE. Ils permettent simplement d'implémenter une règle en testant les valeurs :OLD et :NEW de certaines colonnes. Si une erreur est soulevée, la transaction est automatiquement annulée. Ils peuvent également mettre à jour les valeurs de corrélation (trigger de type BEFORE-ROW INSERT ou UPDATE). Cette manipulation permet de remplacer des valeurs entrant en violation par des données acceptables.

- *Les read triggers :* Ce type de trigger effectue des opérations de lecture sur plusieurs tables. Ces triggers sont généralement utilisés afin d'implémenter des règles qui sont difficilement applicables par les contraintes déclaratives. Ils permettent de mettre en application des règles complexes qui effectuent des lectures non seulement sur la table associée au trigger mais également sur d'autres tables. Dépendant des opérations associées au déclenchement et des relations de la table associée au trigger, des conflits peuvent se produire. Des triggers effectuant des opérations de lecture sur des tables en conflit sont appelés "read-with-conflict triggers". Ces conflits peuvent venir de lectures sur des tables mutantes[5]. Ce type de trigger permet de vérifier la violation de contraintes (généralement complexes).

- *Les write triggers :* Ces triggers sont utilisés afin d'implémenter des règles qui exigent une mise à jour des valeurs de corrélation durant une opération d'insertion ou de mise à jour d'un trigger "Row-Level". Les modifications de données peuvent consister en des opérations d'insertions, de suppressions ou de mises à jour sur des colonnes de la table associée au trigger mais également sur des colonnes d'autres tables. Par ces opérations, d'autres triggers peuvent se déclencher en cascade. Il existe deux versions de write trigger. Les "write-conflict" triggers correspondent à des triggers qui sont en conflit avec les contraintes d'intégrité d'Oracle et les "write-safe" triggers qui ne posent aucune interférence.

Les "read" et "write" triggers sont de complexité non nulle car ils peuvent générer des conflits.

6.5.1 Les tables mutantes

Pour déterminer les situations de conflits, Oracle introduit la notion de table mutante "mutating table". Une table est déclarée "mutante" lorsqu'elle

- est modifiée par le SGBD suite à une opération d'UPDATE, DELETE ou INSERT.
- est lue par le SGBD pour la vérification de l'intégrité référentielle.
- est modifiée par le SGBD pour mettre en œuvre une contrainte de suppression en cascade (ON DELETE CASCADE)

Oracle impose qu'il n'y ait aucune interférence venant d'un trigger lorsqu'une table est déclarée mutante. Toute opération de lecture ou d'écriture sur une table mutante engendre un erreur du type "ORA-4091 "table is mutating". On pourrait véritablement se demander s'il y aura

[5] Cfr la section suivante.

vraiment conflit lorsqu'un read trigger lit dans une table qui est en train d'être lue par le SGBD pour vérifier une contrainte référentielle (vérification des contraintes de clé étrangère ("*update restrict*" et "*delete restrict*" pour la table "parent" et "*insert restrict*" et "*update restrict*" pour la table "enfant"[6])). En fait, il n'y a vraiment conflit que, si un write trigger est en train de modifier une table pendant que le SGBD est en train de lire dans cette même table et inversement. En outre, un autre conflit se produit lorsqu'un write trigger et le SGBD tentent conjointement d'effectuer des opérations d'écriture sur une même table.

Pour mieux analyser ce problème de conflit, la sixième étape a pour but de détecter les éventuels conflits en dégageant, pour chaque trigger "read" ou "write" à implémenter, les tables qui peuvent générer des conflits. Cette analyse se décompose en trois parties.

- Identifier toutes les tables (ascendantes / descendantes) associées à la table sur laquelle sera défini le trigger. En effet, ce sont ces tables qui sont susceptibles de tomber en conflit si une opération d'insertion, de suppression ou de mise à jour est effectuée sur la table associée à la définition du trigger. La solution à ce problème peut être trouvée, d'une part, en analysant le schéma entité-association (s'il existe) et les structures déclarées pour identifier les liens via les clés étrangères ou d'autre part, en interrogeant le dictionnaire des données.

- Sur base de l'ensemble des tables spécifiées par le point précédent, effectuer l'intersection avec l'ensemble des tables définies au sein du corps des procédures stockées (ou des fonctions) appelées par le trigger. Le résultat de cette opération propose une liste des tables susceptibles de générer un conflit.

- Approfondir l'analyse, pour chaque table, pour déterminer si les commandes DML qui déclenche le trigger, engendrent effectivement un conflit. Il y a effectivement conflit sur la table du trigger si cette table se trouve dans l'ensemble résultant de la deuxième partie. Pour les autres tables de l'ensemble, le tableau ci-dessus présente les situations de conflit.

Nous ne considérons que les triggers de type Row-level.

	Foreign key / Event	ON UPDATE RESTRICT	ON INSERT RESTRICT	ON DELETE RESTRICT	ON DELETE CASCADE
Write Trigger on child table writing in parent table	*child INSERT*		Conflict on parent table		
	child DELETE			**(1)**	
	child UPDATE	Conflict on parent table			
Write Trigger on parent table writing in child table	*parent INSERT*		**(2)**		
	parent DELETE			Conflict on child table	
	parent UPDATE	Conflict on child table			
Read Trigger on child table reading in parent table	*child INSERT*		**(3)**		
	child DELETE			**(4)**	
	child UPDATE	**(5)**			
Read Trigger on parent table reading in child table	*parent INSERT*		**(6)**		
	parent DELETE			**(7)**	**(8)**
	parent UPDATE				

[6] Ces notions sont défines en annexe P.114.

1) Il n'y a pas conflit puisque le SGBD n'accède pas à la table "parent" lors d'une suppression d'un enregistrement "enfant".
2) Pas de conflit sur la table "enfant" puisque le SGBD n'accède pas à cette table.
3) Pas de conflit puisque le trigger et le SGBD n'effectuent que des opérations de lecture.
4) Il n'y a pas conflit sur la table "parent" puisque le SGBD n'accède pas à la table "parent" lors d'une suppression d'un enregistrement "enfant".
5) Pas de conflit puisque le trigger et le SGBD n'effectuent que des opérations de lecture.
6) Il n'y a pas conflit sur la table "enfant" puisque le SGBD n'accède pas à la table "enfant" lors d'une suppression d'un enregistrement "parent".
7) Pas de conflit puisque le trigger et le SGBD n'effectuent que des opérations de lecture.
8) Il y a conflit sur la table "enfant" lorsque le SGBD doit supprimer des enregistrements "enfants" par application "DELETE CASCADE".

6.5.2 Résolution du conflit

Considérons une règle quelque peu plus complexe dans notre schéma CLIENTS-TRANSACTIONS-PROD_FINANCIER.

Règle7 : Un client[7] de type particulier (PARTIC) ne peut acheter pour plus de 10 millions par jour.

Pour intégrer cette règle, nous devons effectuer la démarche décrite précédemment à partir de la deuxième étape :

Description de la liste des violations de contraintes (CVL)

Trois cas peuvent se produire et engendrer une violation de la règle :

- Une insertion d'une transaction, à la date du jour et pour un client de type "PARTIC" peut engendrer que la limite des 10 millions soit dépassée pour ce client.
- Une mise à jour de la quantité d'une transaction existante associée à un client de type "PARTIC" et à la date du jour, peut dépasser la limite des 10 millions.
- Le type d'un client peut être mis à jour à la valeur "PARTIC" et peut générer une violation si ce client de type différent de "PARTIC" avait des transactions d'un montant supérieur à 10 millions pour le jour courant.

Le problème de cette situation est que l'on a des triggers sur des tables différentes (un pour la table CLIENTS et un second pour la table TRANSACTIONS pour la même règle). Chaque trigger possède une information partielle (TRA_QUANTITE dans la table transactions et CLI_TYPE dans la table clients). Une jointure via la clé étrangère s'impose pour disposer d'une information complète.

---

[7] Il existe plusieurs types de clients. Le type de clients est enregistré dans le champs CLI_TYPE. Les valeurs de ce champ sont: "PARTIC" pour les clients "particuliers", "SOCPRIV" pour les sociétés privées, "SOCPUB" pour les sociétés publiques, BR pour les clients brokers, "BANQUE" pour les banques, "SOCCO" pour les sociétés commerciales de gestion boursière, "ETRANGER" pour les clients étrangers et "Autres" dans les autres cas.

Description fonctionnelle du trigger (TFD)

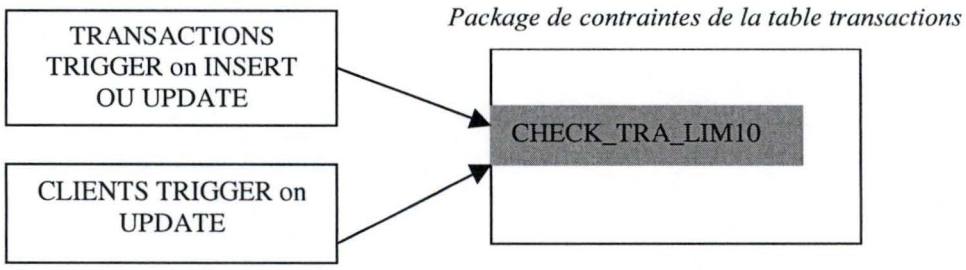
Pour chacun des événements détectés à la phase 2, nous définissons la description fonctionnelle :

- INSERT TRANSACTIONS
  Description : Lors d'une insertion d'une transaction, ce trigger a pour objectif de détecter une violation si un client de type "PARTIC" a un montant de transactions supérieur à 10 millions pour le jour courant. Sur base d'un numéro de client CLI_NUMID, il faut effectuer la somme de ces transactions (TRA_PRIX* TRA_QUANTITE) pour le jour courant.
  Données nécessaires : l'attribut CLI_NUMID sera utilisé pour effectuer la jointure entre les deux tables concernées.
  Type du Trigger : Row Level

- UPDATE TRANSACTIONS
  Description : ce trigger a pour objectif de détecter une violation si pour un client de type "PARTIC", une transaction est mise à jour et que cette opération engendre que le montant des transactions est supérieur à 10 millions pour le jour courant. La méthode de calcul est similaire à celle du trigger.
  Données nécessaires : l'attribut CLI_NUMID sera utilisé pour effectuer la jointure entre les deux tables concernées.
  Type du Trigger : Row Level

- UPDATE CLIENTS
  Description : ce trigger a pour objectif de détecter une violation si le type d'un client est mis à jour à la valeur "PARTIC" et que ce client a un montant de transactions supérieur à 10 millions pour le jour courant. Sur base du numéro de client CLI_NUMID, il faut effectuer la somme de ces transactions (TRA_PRIX* TRA_QUANTITE pour le jour courant.
  Données nécessaires : l'attribut CLI_NUMID sera utilisé pour effectuer la jointure entre les deux tables concernées.
  Type du Trigger : Row Level

Les trois descriptions sont relativement similaires, une seule procédure stockée de nom CHECK_TRA_LIM10 permet d'implémenter la règle.

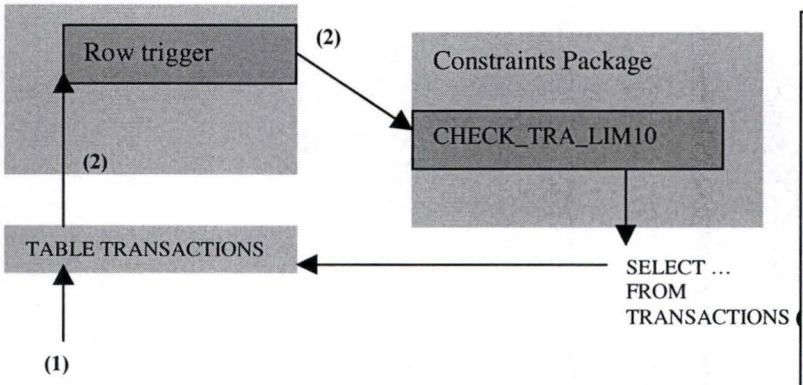


La procédure stockée *CHECK_TRA_LIM10* s'intègre dans le package des contraintes de la table transactions, que nous avons déjà défini.

```
*****************************************************************************************
*                          TRANSACTIONS CONSTRAINTS PACKAGE SPEC. AND BODY              *
*****************************************************************************************
        -- PARTIE SPECIFICATION
        PROCEDURE CHECK_TRA_LIM10[8]
                ( CLIID IN CLIENTS.CLI_NUMID%TYPE);
        -- FIN PARTIE SPECIFICATION
        -- CORPS DE LA PROCEDURE
        PROCEDURE CHECK_TRA_LIM10[9]
                ( CLIID IN CLIENTS.CLI_NUMID%TYPE) IS
                montant NUMBER;
                BEGIN
                SELECT SUM (TRA_PRIX * TRA_QUANTITE) INTO montant
                FROM TRANSACTIONS T, CLIENTS C
                WHERE T.TRA_CLI_NUMID = C.CLI_NUMID
                AND T.TRA_CLI_NUMID = CLIID
                AND C.CLI_TYPE='PARTIC'
                AND T.TRA_DATE = TRUNC(SYSDATE)
                GROUP BY T.TRA_CLI_NUMID;
                IF (montant > 10000000 THEN
                  RAISE_APPLICATION_ERROR (ERREURS.UP_TRA_LIMITE10[10],
                  ERREURS.UP_TRA_LIMITE10_MSG);
                ENDIF;
        END CHECK_TRA_LIM10;
        -- FIN CORPS DE LA PROCEDURE
```

Analyse des conflits éventuels

La sixième étape a pour objectif de déterminer et de résoudre les conflits éventuels. Nous devons définir deux read triggers, l'un associé à la table CLIENTS et l'autre à la table TRANSACTIONS. Nous allons, d'abord, analysé le trigger associé à la table TRANSACTIONS. Comme nous l'avons précisé précédemment, cette analyse se subdivise en trois étapes. Les tables résultant de la première sous-étape forment l'ensemble [CLIENTS-TRANSACTIONS-PRODUIT]. La deuxième étape réduit cet ensemble [CLIENTS-TRANSACTIONS] . La troisième permet de diagnostiquer automatiquement un conflit sur la table [TRANSACTIONS].



(1) Une opération d'UPDATE est effectuée par l'application. Le SGBD prend cette commande en charge et accède en écriture à la table TRANSACTIONS.
(2) Le trigger est déclenché et appelle la procédure stockée spécifiée dans le package des contraintes en passant les paramètres (valeurs de corrélation si nécessaire).
(3) La procédure exécute une instruction SELECT sur la table TRANSACTIONS qui cause une erreur de "table mutante".

[8] Cette partie fournit la spécification de la procédure. Elle doit être insérée dans la partie spécification du package associé aux transactions.

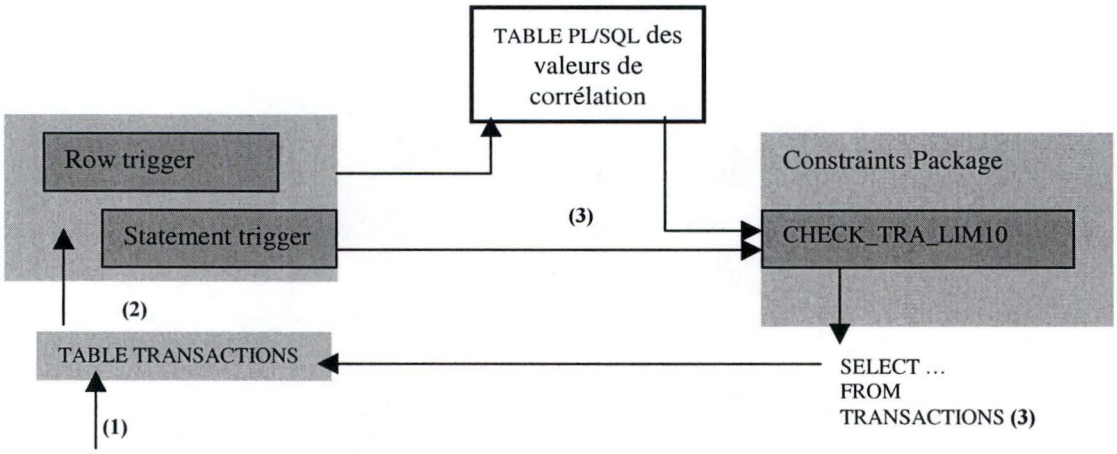
[9] Cette partie fournit le corps de la procédure. Il doit être inséré dans la partie BODY du package associé aux transactions.

[10] Le message d'erreur suivant a été inséré dans le package des erreurs.
Partie spécification: UP_TRA_LIMITE10 CONSTANT INTEGER := -20004;
Corps du message: UP_TRA_LIMITE10_MSG VARCHAR2(88) NOT NULL := 'IMPOSSIBLE D'ENREGISTRER LES CHANGEMENTS, UN PARTICULIER NE PEUT AVOIR PLUS DE 10 MILLIONS DE TRANSACTIONS PAR JOUR';

Pour la table CLIENTS, le trigger associé à la table TRANSACTIONS nous place dans le cas où soit, un read trigger on child table, reading in parent table, child UPDATE, soit, un read trigger, on child table, reading in parent table, child INSERT.
En examinant le tableau d'occurrences des conflits, on remarque qu'il n'y aura pas de conflit pour la table CLIENTS, quelle que soit la clause de clé étrangère que le système doit faire respecter.

Résolution du conflit

Pour résoudre le conflit de table mutante causé par le trigger "row-level" sur TRANSACTIONS, Oracle propose de le substituer par un trigger "statement-level" Cette opération exige l'utilisation d'une table temporaire PL/SQL qui enregistrera toutes les lignes cibles de l'UPDATE ou l'INSERT. Le trigger row-level ne fera plus appel à la procédure CHECK_TRA_LIM10 mais sa mission sera de placer les valeurs de corrélation dans la table temporaire suite à l'exécution de la commande DML. Pour implémenter cette technique, un trigger statement-level sera créé. Il appellera la procédure CHECK_TRA_LIM10 pour chaque numéro de clients stocké dans la table temporaire.
L'avantage est que l'implémentation n'exige aucun changement dans l'écriture du package des contraintes (C'est la raison pour laquelle cette étape succède l'écriture du package). Cependant, cette technique exige de créer la table temporaire PL/SQL dans un package. Il faut donc écrire, et d'une part, un before-statement-level trigger qui initialise la table temporaire, d'autre part, un trigger de type row-level qui garnit la table temporaire avec les valeurs de corrélation et enfin, un after-statement-level trigger qui appèle la procédure CHECK_TRA_LIM10 pour chacune des valeurs stockées dans la table temporaire.
Généralement, la table temporaire stocke les valeurs des clés primaires ou étrangères. Dans notre cas, elle enregistre les numéros de clients des enregistrements de TRANSACTIONS mis à jour ou insérés. Nous pouvons schématiser ce mécanisme :



(1) Une opération d'UPDATE est effectuée par l'application. Le SGBD prend cette commande en charge et accède en écriture à la table TRANSACTIONS.
(2) Le trigger row-level sauve la clé primaire ou la clé étrangère pour chaque enregistrement mis à jour.
(3) Le trigger statement-level appelle la procédure CHECK_TRA_LIM10 qui met en œuvre la contrainte pour chaque clé primaire de la table PL/SQL.
(4) La table n'est plus mutante et par conséquent, le trigger statement-level peut effectuer une lecture sur la table transactions.

Résolution du conflit " table mutanting"

La déclaration de la table temporaire est réalisée par un package PL/SQL. La spécification du package spécifie les opérations d'insertion, de recherche. Il existe deux compteurs d'indexation. Le compteur IDX est utilisé pour les insertions au sein de la table, il ne peut jamais décroître et sa valeur correspond à la taille de la table. Le compteur ITERATOR est utilisé pour parcourir la table. La procédure SET_ITERATOR doit être appelée avant d'effectuer toutes opérations de lecture. Définissons ce package que nous appellerons CLI_ID_TABLE_PKG.

```
/*****************************************************************************
*                        PL/SQL TABLE PACKAGE SPEC. AND BODY POUR CLI_ID                    *
*****************************************************************************/
CREATE OR REPLACE PACKAGE CLI_ID_TABLE_PKG IS
        PROCEDURE CLEAR[11];
        PROCEDURE SET_ITERATOR;
        PROCEDURE PUT (CLI_ID IN CLIENTS.CLI_NUMID%TYPE);
        FUNCTION NEXT_VAL RETURN CLIENTS.CLI_NUMID%TYPE[12] ;
        FUNCTION  MORE_IN_TABLE RETURN BOOLEAN[13] ;
        EXCEPTION UNDERFLOW;
END CLI_ID_TABLE_PKG;

CREATE OR REPLACE PACKAGE BODY CLI_ID_TABLE_PKG IS
        TYPE TABLE_TYPE IS TABLE OF CLIENTS.CLI_NUMID%TYPE
        INDEXED BY BINARY_INTEGER;
        TAB TABLE_TYPE;
        IDX BINARY_INTEGER :=0;
        ITERATOR BINARY_INTEGER :=0;
        PROCEDURE CLEAR IS
        BEGIN  IDX :=0; END CLEAR;
        PROCEDURE SET_ITERATOR IS
        BEGIN  ITERATOR := IDX; END SET_ITERATOR;
        PROCEDURE PUT (CLI_ID IN CLIENTS.CLI_NUMID%TYPE) IS
        BEGIN
                IDX := IDX + 1;
                TAB(IDX) := CLI_ID;
        END PUT;
        FUNCTION  NEXT_VAL RETURN CLIENTS.CLI_NUMID%TYPE IS
        BEGIN
                ITERATOR := ITERATOR -1;
                RETURN TAB(ITERATOR+1);
        EXCEPTION
                WHEN NO_DATA_FOUND THEN RAISE UNDERFLOW;
        END NEXT_VAL;
        FUNCTION  MORE_IN_TABLE RETURN BOOLEAN IS
        BEGIN
                RETURN (ITERATOR >0 );
        END MORE_IN_TABLE;
END CLI_ID_TABLE_PKG;
```

Une fois le package défini, celui-ci peut être tout à fait utilisé par tout trigger qui nécessite l'utilisation d'une table temporaire associée au numéro de client (CLI_NUMID). Pour chaque instruction SQL d'insertion, de suppression ou de mise à jour qui exige l'implémentation de contraintes procédurales, la table PL/SQL doit être initialisée pour enregistrer les valeurs de corrélation venant du trigger row-level. La procédure CLEAR devra être appelée avant toute exécution du trigger statement-level, soit par un before-statement-trigger. Un after-statement-trigger fera, quant à lui, appel à la procédure de vérification qui itérera, pour chaque valeur de

[11] La procédure CLEAR doit être appelée avant toute utilisation de la table temporaire.
[12] La fonction NEXT_VAL soulève une exception si l'appelant retourne une valeur en deçà de la première valeur du tableau.
[13] Cette fonction effectue un contrôle de l'itération.

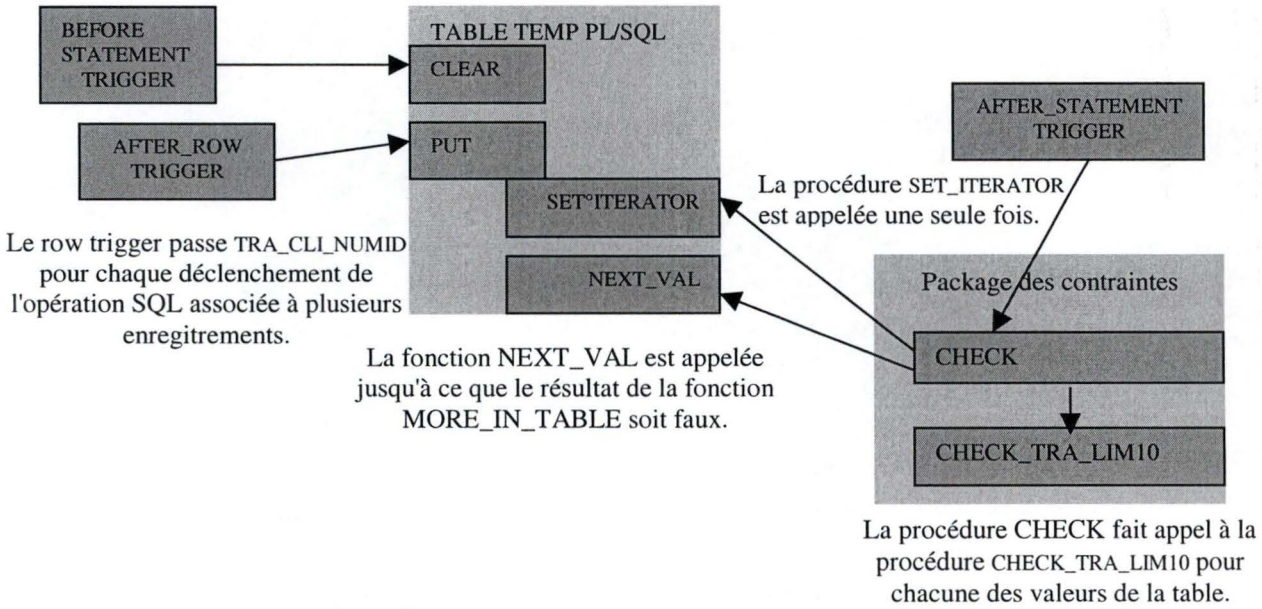
la table PL/SQL grâce aux procédures NEXT_VAL et MORE_IN_TABLE, l'appel à la procédure CHECK_TRA_LIM10. Cette procédure de vérification (CHECK) est une nouvelle procédure à insérer dans le package de contraintes (TRANSACTIONS_CONS_PKG).

L'implémentation de la procédure CHECK est la suivante :

```
*****************************************************************************************
*                          TRANSACTIONS CONSTRAINTS PACKAGE SPEC. AND BODY              *
*****************************************************************************************
        -- PARTIE SPECIFICATION
        PROCEDURE CHECK;
        -- FIN PARTIE SPECIFICATION

        -- CORPS DE LA PROCEDURE
        PROCEDURE CHECK IS
        BEGIN
                CLI_ID_TABLE_PKG.SET_ITERATOR;
                WHILE CLI_ID_TABLE_PKG.MORE_IN_TABLE LOOP
                        CHECK_TRA_LIM10(CLI_ID_TABLE_PKG.NEXT_VAL);
                END LOOP;
        END CHECK;
        FIN CORPS DE LA PROCEDURE
```

En résumé, il faut créer le package associé à la gestion de la table PL/SQL, modifier le package des contraintes, insérer la procédure CHECK, créer un before-statement trigger pour initialiser la table PL/SQL, modifier le row-level trigger pour enregistrer une valeur unique de clé dans la table et enfin, créer un after-statement trigger qui appliquera la contrainte sur tous les enregistrements affectés.
Le schéma ci-dessous permet de mieux visualiser le mécanisme de correction de conflit par la création d'une table temporaire PL/SQL.



Interactions entre les triggers, le package de la table temporaire PL/SQL et le package des contraintes

Triggers associés à cette règle

Les triggers doivent être encore implémentés. Voici les différents triggers qui permettent d'implémenter cette règle 7 sans qu'en résultent des messages d'erreurs du type "table

mutating". Le raisonnement que nous avons développé doit être également réalisé pour la table CLIENTS. Cependant, il est relativement proche puisqu'il nécessite l'utilisation d'une table temporaire pour contrecarrer les problèmes de conflits. Nous nous limitons à l'écriture des triggers associés.

Triggers associés à la table TRANSACTIONS

- Pour toute insertion dans la table TRANSACTIONS

```
CREATE OR REPLACE TRIGGER TRANSACTIONS_BIS
BEFORE INSERT ON TRANSACTIONS
BEGIN
        CLI_ID_TABLE_PKG.CLEAR;
END;

CREATE OR REPLACE TRIGGER TRANSACTIONS_AIR
AFTER INSERT ON TRANSACTIONS
FOR EACH ROW
BEGIN
        CLI_ID_TABLE_PKG.PUT(:NEW.TRA_CLI_NUMID);
END;

CREATE OR REPLACE TRIGGER TRANSACTIONS_AIS
AFTER INSERT ON TRANSACTIONS
BEGIN
        TRANSACTIONS_CONS_PKG.CHECK;
END;
```

- Pour toute mise à jour dans la table TRANSACTIONS

```
CREATE OR REPLACE TRIGGER TRANSACTIONS_BUS
BEFORE UPDATE ON TRANSACTIONS
BEGIN
        CLI_ID_TABLE_PKG.CLEAR;
END;

CREATE OR REPLACE TRIGGER TRANSACTIONS_AUR
AFTER UPDATE ON TRANSACTIONS
FOR EACH ROW
BEGIN
        CLI_ID_TABLE_PKG.PUT(:NEW.TRA_CLI_NUMID);
END;

CREATE OR REPLACE TRIGGER TRANSACTIONS_AUS
AFTER UPDATE ON TRANSACTIONS
BEGIN
        TRANSACTIONS_CONS_PKG.CHECK;
END;
```

Triggers associés à la table CLIENTS

- Pour toute modification dans la table CLIENTS

```
CREATE OR REPLACE TRIGGER CLIENTS _BUS
BEFORE UPDATE ON CLIENTS
BEGIN
        CLI_ID_TABLE_PKG.CLEAR;
END;

CREATE OR REPLACE TRIGGER CLIENTS _AUR
```

```
AFTER UPDATE ON CLIENTS
FOR EACH ROW
BEGIN
        CLI_ID_TABLE_PKG.PUT(:NEW.CLI_NUMID);
END;

CREATE OR REPLACE TRIGGER CLIENTS _AUS
AFTER UPDATE ON CLIENTS
BEGIN
        TRANSACTIONS_CONS_PKG.CHECK;
END;
```

Ces triggers qui sont encodés dans les packages respectifs TRANSACTIONS_TRG et CLIENTS_TRG, doivent être testés afin confirmer les effets de la règle spécifiée. Au besoin, certaines corrections doivent être opérées.

Au niveau de la maintenance des règles, on peut remarquer que la règle est principalement définie au sein de la procédure stockée CHECK_TRA_LIM10. La gestion des règles est donc directement adaptable et applicable par la modification de la procédure. L'implémentation de la règle a exigé la création de deux packages (TRANSACTIONS_CONS_PKG et CLI_ID_TABLE_PKG) et l'écriture de neuf triggers. Cette implémentation est relativement complexe.

Au niveau du trigger row-level, d'autres implémentations de règles peuvent exiger le stockage dans la table temporaire PL/SQL, de plus d'informations que la clé primaire. Certains cas exigent la sauvegarde d'une clé concaténée, d'autres imposent le stockage des anciennes et nouvelles valeurs de corrélation, enfin certaines situations imposent le sauvetage des enregistrements dans leur entièreté.

Le sauvetage de la clé concaténée est nécessaire dans la table PL/SQL lorsque la violation d'une contrainte référence une colonne d'une clé concaténée. La clé complète doit être enregistrée parce que le after-statement trigger doit être capable d'identifier l'enregistrement qui est concerné par l'instruction SQL. Pour rappel, une table qui possède une clé concaténée, peut être une solution pour résoudre des associations MANY-TO-MANY.

L'enregistrement des valeurs de corrélation est exigé dans la table temporaire lorsque des comparaisons des valeurs de corrélation :OLD et :NEW doivent être effectuées pour implémenter la règle.

6.6 Utilisation des triggers et des procédures stockées

On peut remarquer que, si les principes des triggers et des procédures stockées sont relativement simples à comprendre, leur implémentation peut rapidement devenir complexe.

Cette tendance se confirme à travers diverses statistiques[14] d'utilisation des triggers au sein des environnements Oracle puisqu'en général, plus de 50% des utilisateurs n'utilisaient pas les triggers dans leur SGBD. On voit également que les règles mises en application dans les SGDB étaient des règles relativement simples et associées à des utilisations de sécurité et de contrôle des accès. Cependant, par les nouveaux développements du langage PL/SQL et des outils de développement, la tendance s'inverse et actuellement, on peut remarquer une généralisation beaucoup plus importante et plus diversifiée de leur utilisation (implémentation de règles stratégiques complexes, audit, statistiques, duplication, gestion de bases distribuées et (ou)

[14] Statistiques de mai 1998

temporelles,...). En outre, les domaines d'application des triggers se sont largement étendus puisque les triggers et les procédures stockées sont utilisés dans les domaines de pointe tels que la gestion des réseaux, le contrôle de qualité et de sécurité dans les processus industriels et la gestion des systèmes d'information. Une meilleure connaissance théorique et empirique de leurs fonctions et de leurs utilisations permet actuellement de mieux les valoriser vis-à-vis des développeurs. Gageons que la technique des triggers et des procédures stockées permettra d'apporter un réel potentiel au SGDB en leur permettant de développer des contrôles actifs garantissant une performance optimale et une meilleure rentabilité des systèmes.

Le dernier chapitre de notre travail va présenter d'autres modélisations de règles qui nécessitent des techniques d'implémentation particulières ou du moins, certains subtilités d'écriture (les flags, par exemple).

Chapitre 7:

Implémentation des triggers: Cas particuliers

7.1 Les triggers récursifs

7.1.1 Liste de vérification de contrainte (Constraints Enforcement List CEL)

7.1.2 Description fonctionnelle du trigger

7.1.3 Implémentation de la règle

7.2 Les triggers durant des opérations de DELETE CASCADE

7.2.1 Application de la règle uniquement sur la table TRANSACTIONS

7.2.2 Application de la règle dans tous les cas de DELETE CASCADE

Chapitre 7 :.

Implémentation des triggers : Cas particuliers

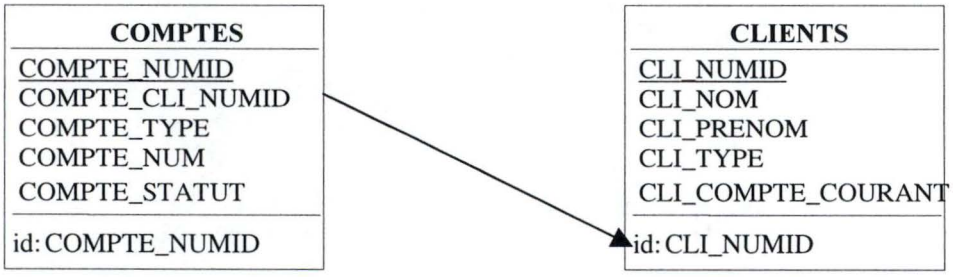
Nous poursuivons notre analyse par l'implémentation de triggers pouvant intervenir dans des situations particulières. Leurs implémentations peuvent être relativement complexes mais des subtilités d'écriture peuvent simplifier leur développement.

7.1 Les triggers récursifs

Les triggers récursifs sont, par définition, des triggers modifiant la table sur laquelle est défini le trigger déclencleur. Cette définition doit inclure les triggers d'insertion qui font appel à des procédures stockées contenant des instructions d'insertion sur la table qui a déclenché le trigger. Il est également possible d'effectuer des combinaisons de ces éléments: imaginons un trigger d'insertion qui fait appel une procédures stockée déclenchant, elle-même, une instruction d'insertion sur la même table.

Il est évidemment préférable d'éviter ce type de trigger car les déclenchements successifs peuvent engendrer des cycles de survenances. Bien qu'il existe des techniques pour interrompre ces cycles, le résultat de ces opérations peut poser quelques problèmes de cohérence et de performance. Cependant, leur implémentation est réalisable à travers la technique des "flags".

Nous allons compléter notre schéma CLIENTS-TRANSACTIONS-PROD_FINANCIER en ajoutant une quatrième table regroupant tous les comptes des clients à l'exception du compte courant qui est directement accessible dans la table CLIENTS. Cette table COMPTES est composée respectivement d'une clé primaire (COMPTE_NUMID, clé technique), d'une clé étrangère vers le la clé primaire de CLIENTS (COMPTE_CLI_NUMID), du type de compte (COMPTE_TYPE de type "normal" ou "privilegie[1]"), du numéro de compte (COMPTE_NUM[2]) et enfin, du statut du compte (COMPTE_STATUT : "ouvert", "ferme"). Le schéma ci-dessous décrit cette nouvelle situation.



Nous désirons insérer la règle suivante :

Règle 8 : un client qui ouvre un compte de type privilégié, a automatiquement l'ouverture d'un compte de type normal. Cette règle est nécessaire car certaines opérations ne s'effectuent que sur des comptes de type "normal".

---

[1] Les comptes de type privilégié sont réservés aux clients effectuant des opérations conséquentes (en général, plus de 20.000.000 francs par an).

[2] Le numéro de compte ne peut être utilisé comme clé primaire car un client peut avoir plusieurs comptes associés à un même numéro de compte.

On peut constater que cette règle va faire intervenir un trigger récursif. Pour implémenter ce trigger, nous allons effectuer les différentes étapes proposées par la méthodologie exposée au chapitre précédent.

7.1.1 Liste de vérification de contraintes (Constraints Enforcement List - CEL)

Cette étape correspond, soit à l'identification des cas où la règle peut être violée, soit à la mise en évidence des cas d'occurrence de la règle. Un seul cas peut être se produire : Une insertion dans la table COMPTES d'un compte de type "privilegie" exige une insertion récursive dans la table COMPTES d'un enregistrement "normal". Nous supposons que la mise à jour du type de compte de la table COMPTES est interdite.

7.1.2 Description fonctionnelle du trigger

INSERT COMPTES

Description : si COMPTE_TYPE="privilegie", alors, de manière récursive, un INSERT de même COMPTE_CLI_NUMID et COMPTE_TYPE ="normal" doit s'effectuer. Cette insertion récursive doit se faire au cours du traitement d'un trigger after-statement. Cependant, un flag doit être utilisé pour permettre de bloquer la logique d'exécution du trigger d'insertion pendant l'insertion récursive. En outre, l'utilisation d'une table temporaire est requise. Pour des raisons d'utilisations ultérieures (généralisation d'autres règles), la table temporaire PL/SQL va sauvegarder plus d'informations que l'exige notre cas.

Données nécessaires : COMPTE_CLI_NUMID

Type du Trigger : Statement Level

7.1.3 Implémentation de la règle

```
*************************************************************************************/
*       PL/SQL TABLE PACKAGE SPEC. AND BODY POUR COMPTE.CLI_NUMID        *
*************************************************************************************/
CREATE OR REPLACE PACKAGE COMPTE _ID_TABLE_PKG IS
        TYPE COMPTE_REC_TYPE (
                COMPTE_CLI_NUMID  COMPTE.CLI_NUMID%TYPE,
                OLD_COMPTE_NUMID          COMPTE_NUMID%TYPE,
                OLD_COMPTE_TYPE           COMPTE_TYPE%TYPE,
                NEW_COMPTE_NUMID          COMPTE_NUMID%TYPE,
                NEW_COMPTE_TYPE           COMPTE_TYPE%TYPE,
                NEW_COMPTE_NUM            COMPTE_NUM%TYPE,
                NEW_COMPTE_STATUT         COMPTE_STATUT%TYPE;
        PROCEDURE CLEAR_COMPTE_TABLE;
        PROCEDURE SET_ITERATOR;
        FUNCTION NEXT_REC RETURN COMPTE_REC_TYPE;
        FUNCTION  MORE_IN_TABLE RETURN BOOLEAN;
        PROCEDURE INSERT_VALEUR (
                COMPTE_CLI_NUMID  COMPTE.CLI_NUMID%TYPE,
                OLD_COMPTE_NUMID          COMPTE_NUMID%TYPE,
                OLD_COMPTE_TYPE   COMPTE_TYPE%TYPE,
                NEW_COMPTE_NUMID          COMPTE_NUMID%TYPE,
                NEW_COMPTE_TYPE   COMPTE_TYPE%TYPE,
                NEW_COMPTE_NUM    COMPTE_NUM%TYPE,
```

```
                NEW_COMPTE_STATUT        COMPTE_STATUT%TYPE);
END COMPTE.CLI_NUMID;
CREATE OR REPLACE PACKAGE BODY COMPTE _ID_TABLE_PKG IS
        TYPE
        COMPTE_CLI_NUMID_TAB_TYPE      IS TABLE OF   COMPTE.CLI_NUMID%TYPE
                                                     INDEX BY BINARY_INTEGER;
        COMPTE_NUMID_TAB_TYPE          IS TABLE OF   COMPTE_NUMID%TYPE
                                                     INDEX BY BINARY_INTEGER;
        COMPTE_TYPE_TAB_TYPE           IS TABLE OF   COMPTE_TYPE%TYPE
                                                     INDEX BY BINARY_INTEGER;
        COMPTE_NUM_TAB_TYPE            IS TABLE OF   COMPTE_NUM%TYPE
                                                     INDEX BY BINARY_INTEGER;
        COMPTE_STATUT_TAB_TYPE IS TABLE OF   COMPTE_STATUT%TYPE
                                                     INDEX BY BINARY_INTEGER;

COMPTE_CLI_NUMID_TAB            COMPTE_CLI_NUMID_TAB_TYPE;
OLD_COMPTE_NUMID_TAB            COMPTE_NUMID_TAB_TYPE;
OLD_COMPTE_TYPE_TAB             COMPTE_TYPE_TAB_TYPE;
NEW_COMPTE_NUMID_TAB            COMPTE_NUMID_TAB_TYPE;
NEW_COMPTE_TYPE_TAB             COMPTE_TYPE_TAB_TYPE;
NEW_COMPTE_NUM_TAB              COMPTE_NUM_TAB_TYPE;
NEW_COMPTE_STATUT_TAB           COMPTE_STATUT_TAB_TYPE;
tab_compteur                    NUMBER;
tab_iterator                    NUMBER :=0 ;

        PROCEDURE CLEAR_COMPTE_TABLE IS
        BEGIN  tab_compteur:=0;          END CLEAR_COMPTE_TABLE;

        PROCEDURE SET_ITERATOR IS
        BEGIN tab_iterator:= tab_compteur; END SET_ITERATOR;

        FUNCTION NEXT_REC RETURN COMPTE_REC_TYPE IS
        REC COMPTE_REC_TYPE;
        BEGIN
                REC.COMPTE_CLI_NUMID     := COMPTE_CLI_NUMID_TAB(tab_iterator);
                REC.OLD_COMPTE_NUMID     := OLD_COMPTE_NUMID_TAB(tab_iterator);
                REC.OLD_COMPTE_TYPE      := OLD_COMPTE_TYPE_TAB(tab_iterator);
                REC.NEW_COMPTE_NUMID     := NEW_COMPTE_NUMID_TAB(tab_iterator);
                REC.NEW_COMPTE_TYPE      := NEW_COMPTE_TYPE_TAB(tab_iterator);
                REC.NEW_COMPTE_NUM       := NEW_COMPTE_NUM_TAB(tab_iterator);
                REC.NEW_COMPTE_STATUT    := NEW_COMPTE_STATUT_TAB(tab_iterator);
                tab_iterator := tab_iterator - 1;
                RETURN REC;
        END NEXT_REC;

        FUNCTION  MORE_IN_TABLE RETURN BOOLEAN IS
        BEGIN
                RETURN (tab_iterator >0 );
        END MORE_IN_TABLE;
        PROCEDURE INSERT_VALEUR (
                COMPTE_CLI_NUMID  COMPTE.CLI_NUMID%TYPE,
                OLD_COMPTE_NUMID          COMPTE_NUMID%TYPE,
                OLD_COMPTE_TYPE           COMPTE_TYPE%TYPE,
                NEW_COMPTE_NUMID          COMPTE_NUMID%TYPE,
                NEW_COMPTE_TYPE           COMPTE_TYPE%TYPE,
                NEW_COMPTE_NUM            COMPTE_NUM%TYPE,
                NEW_COMPTE_STATUT         COMPTE_STATUT%TYPE) IS
        BEGIN
                tab_compteur := tab_compteur +1;
                COMPTE_CLI_NUMID_TAB(tab_compteur)      := COMPTE_CLI_NUMID;
                OLD_COMPTE_NUMID_TAB(tab_compteur)      := OLD_COMPTE_NUMID;
                OLD_COMPTE_TYPE_TAB(tab_compteur)       := OLD_COMPTE_TYPE;
```

```
                NEW_COMPTE_NUMID_TAB(tab_compteur)      := NEW_COMPTE_NUMID;
                NEW_COMPTE_TYPE_TAB(tab_compteur)       := NEW_COMPTE_TYPE;
                NEW_COMPTE_NUM_TAB(tab_compteur)        := NEW_COMPTE_NUM;
                NEW_COMPTE_STATUT_TAB(tab_compteur):= NEW_COMPTE_STATUT;
        END INSERT_VALEUR;
END COMPTE _ID_TABLE_PKG;

/*****************************************************************************************
*                    CONSTRAINT PACKAGE SPEC. AND BODY COMPTES_CONS_PKG       *
*****************************************************************************************/

CREATE OR REPLACE PACKAGE COMPTES_CONS_PKG IS
        FLAG_REGLE8 BOOLEAN := FALSE; --GLOBAL FLAG
        PROCEDURE INSERTION_REGLE8 (REC COMPTE _ID_TABLE_PKG.COMPTE_REC_TYPE);
END COMPTES_CONS_PKG;

CREATE OR REPLACE PACKAGE BODY COMPTES_CONS_PKG IS
PROCEDURE INSERTION _REGLE8 (REC COMPTE _ID_TABLE_PKG.COMPTE_REC_TYPE) IS
        BEGIN
                IF ( REC.NEW_COMPTE_TYPE=' privilegie") THEN
                -- INSERTION DU COMPTE DE TYPE 'normal'.
                INSERT INTO COMPTES VALUES (COMPTE_NUMID.NEXTVAL,
                REC.COMPTE_CLI_NUMID, 'normal', REC.NEW_COMPTE_NUM,
                REC.NEW_COMPTE_STATUT);
                END IF;
        END INSERTION _REGLE8;
END COMPTES_CONS_PKG;


/*****************************************************************************************
*                                        ECRITURE DES TRIGGERS                          *
*****************************************************************************************/
        CREATE OR REPLACE TRIGGER COMPTES_BIS
                BEFORE INSERT ON COMPTES
        BEGIN
                IF NOT COMPTES_CONS_PKG.FLAG_REGLE8 THEN
                                COMPTE _ID_TABLE_PKG.CLEAR_COMPTE_TABLE;
                END IF;
        END;

        CREATE OR REPLACE TRIGGER COMPTES_AIR
        AFTER INSERT ON COMPTES
        FOR EACH ROW
        BEGIN
                IF NOT COMPTES_CONS_PKG.FLAG_REGLE8 THEN
                   COMPTES_CONS_PKG.INSERT_VALEUR (:NEW.COMPTE_CLI_NUMID,
                        :OLD.COMPTE_NUMID,:OLD.COMPTE_TYPE,:NEWCOMPTE_NUMID,
                        :NEW.COMPTE_TYPE,:NEW.COMPTE_NUM, :NEW.COMPTE_STATUT);
                END IF;
        END;

        CREATE OR REPLACE TRIGGER COMPTES_AIS
        AFTER INSERT ON COMPTES
        BEGIN
            IF NOT COMPTES_CONS_PKG.FLAG_REGLE8 THEN
-- Fixation du flag de manière à ce que le trigger ne se déclenche pas durant l'opération récursive.
                COMPTES_CONS_PKG.FLAG_REGLE8 := true;
                COMPTES_CONS_PKG.SET_ITERATOR;
                WHILE COMPTE _ID_TABLE_PKG.MORE_IN_TABLE LOOP
                  --appel à la procédure d'insertion d'un compte de type "normal"
                  COMPTES_CONS_PKG.INSERTION_REGLE8(COMPTE.CLI_NUMID.NEXT_REC);
                END LOOP;
                COMPTES_CONS_PKG.FLAG_REGLE8 := false;
```

```
        END IF;
END;
```

Ce mécanisme de blocage combiné à l'utilisation d'une table temporaire permet de résoudre le problème associé à l'opération d'insertion récursive.

7.2 Les triggers durant des opérations de DELETE CASCADE

Pour rappel, une table est déclarée mutante si elle est en train d'être mise à jour conformément à une contrainte de DELETE CASCADE. Une stratégie d'implémentation de triggers nécessite une considération particulière lorsque ceux-ci peuvent se déclencher durant une opération de suppression de "DELETE CASCADE". Nous pouvons présenter une stratégie d'implémentation :

- Créer un table temporaire PL/SQL permettant de sauvegarder les données pertinentes provenant d'un trigger row-level. Cette démarche est semblable aux tables temporaires que nous avons déjà créées.
- Créer un package de contraintes comprenant les procédures d'implémentation des contraintes à modéliser (en tenant compte des arguments).
- Ecrire un trigger statement-level qui initialise la table PL/SQL.
- Ecrire un trigger row-level qui remplit la table temporaire.
- Ecrire un trigger after-statement qui applique la règle à chaque enregistrement de la table temporaire.
- Composer un package qui assure la gestion des flags (initialisation et gestion). Ces flags sont lus par les triggers "enfant" pour déterminer l'événement qui invoque le trigger "delete child" (un delete cascade ou une suppression sur la table).
- Rédiger, en utilisant le package des contraintes, les triggers statement-level et row-level sur la table "parent" qui mettent en application la règle désirée.

Modélisons la règle suivante :
Soit la règle 9 : *une transaction de statut (TRA_STATUT) "cloturee" ne peut être supprimée si le type de client (CLI_TYPE) associé à la transaction est "*ETRANGER"

Cette règle va générer les différents composants :

1. Un package PL/SQL associé à la table TRANSACTIONS. Le nom de ce package sera TRANSACTION_TABLE_PKG.
2. Un trigger before-delete statement : il permettra d'initialiser la table PL/SQL.
3. Un trigger after-delete-row-level : celui-ci permettra d'enregistrer les numéros de transaction TRA_NUMID dans la table temporaire en utilisant la procédure PUT.
4. Un trigger after-delete-statement-level : la logique de ce trigger est basée sur l'interprétation de la règle.

Si on examine le schéma relationnel de notre cas, on peut voir que la table TRANSACTIONS est une table "enfant" par rapport aux tables CLIENTS et PROD_FINANCIER. Il existe plusieurs manières d'appliquer la règle durant une opération de DELETE CASCADE.

- La règle est appliquée seulement pour des opérations de suppression dans la table TRANSACTIONS. Cela signifie que, lorsqu'un client est supprimé, la suppression peut se propager sur la table TRANSACTIONS et supprimer des enregistrements sans se soucier de la règle. Il en est de même pour une suppression dans la table

PROD_FINANCIER. La règle n'est donc pas d'application lorsqu'un enregistrement "parent" est supprimé.

- Une deuxième possibilité est d'appliquer la règle en toutes circonstances. Si un client étranger est supprimé et que ce dernier possède des transactions clôturées, un message d'erreur est affiché et la transaction est annulée. Si un produit financier est supprimé et que des transactions associées à ce produit et à un client de type ETRANGER sont clôturées, une exception est générée. La transaction est également annulée.
- La règle peut s'appliquer seulement dans certains cas de DELETE CASCADE. Par exemple, la règle ne s'applique que dans le cas de suppression sur les tables TRANSACTIONS et CLIENTS mais pas dans ceux sur la table PROD_FINANCIER.

7.2.1 Application de la règle uniquement sur la table TRANSACTIONS

La règle n'est pas d'application pour les suppressions provenant d'une opération de DELETE CASCADE conséquente à une suppression sur la table "parent". La règle n'est appliquée que pour les suppressions directes sur la table TRANSACTIONS.

Deux stratégies peuvent modéliser cette situation :

1. Il suffit de fixer, dans un package de spécification, des flags globaux qui permettent d'identifier les tables en cours de modification. Cette technique exige, par conséquent, la création d'un trigger before-delete-statement sur chacune des tables "parent" pour fixer le flag associé et un trigger after-statement sur la table TRANSACTIONS qui va lire le flag. En fonction de la valeur de cette variable, il est possible de déterminer si la suppression s'établit dans le cadre d'une opération DELETE CASCADE. Notons que si l'opération de DELETE CASCADE s'établit sur plusieurs niveaux, les flags doivent être fixés et relâchés à chaque niveau.
2. Une deuxième stratégie est de détecter les erreurs de type "table mutating", de les considérer comme une exception et de les ignorer.

Pour appliquer la même règle, les deux stratégies sont relativement différentes. Cependant, la première offre une plus grande flexibilité car elle permet de personnaliser les opérations de DELETE CASCADE en fonction de la table "parent" associée. Ce souci de flexibilité est toujours un élément important pour la maintenance des règles. La deuxième solution ne se préoccupe nullement de l'origine de l'opération de suppression.

Nous pouvons implémenter ces deux stratégies uniquement par rapport aux deux tables "parent".

1) Stratégie utilisant les flags: les suppressions provenant d'une opération de suppression associée à un DELETE CASCADE sont ignorées. L'implémentation des triggers associés à la table "enfant" n'est pas étudiée puisque nous avons déjà développé ce type de trigger.

```
/*****************************************************************************
*                          ECRITURE DU PACKAGE DES FLAGS                     *
*****************************************************************************/
    CREATE OR REPLACE PACKAGE DELETE_CASCADE_FLAG_PKG IS
        CLIENTS_TAB_FLAG BOOLEAN :=FALSE;
        PROD_FINANCIER_TAB_FLAG BOOLEAN :=FALSE;
    END DELETE_CASCADE_FLAG_PKG;
```

```
/*****************************************************************************************
*                    ECRITURE DES TRIGGERS ASSOCIES AU FLAG DES TABLES "PARENT"          *
*****************************************************************************************/
        CREATE OR REPLACE TRIGGER CLIENTS_BDS
        BEFORE DELETE ON CLIENTS
        BEGIN
                DELETE_CASCADE_FLAG_PKG.PROD_FINANCIER_TAB_FLAG:=TRUE;
        END;
        CREATE OR REPLACE TRIGGER CLIENTS_ADS
        AFTER DELETE ON CLIENTS
        BEGIN
                DELETE_CASCADE_FLAG_PKG.PROD_FINANCIER_TAB_FLAG:=FALSE;
        END;

        CREATE OR REPLACE TRIGGER PROD_FINANCIER_BDS
        BEFORE DELETE ON PROD_FINANCIER
        BEGIN
                DELETE_CASCADE_FLAG_PKG.PROD_FINANCIER_TAB_FLAG:=TRUE;
        END;
        CREATE OR REPLACE TRIGGER PROD_FINANCIER_ADS
        AFTER DELETE ON PROD_FINANCIER
        BEGIN
                DELETE_CASCADE_FLAG_PKG.PROD_FINANCIER_TAB_FLAG:=FALSE;
        END;
/*****************************************************************************************
*       ECRITURE DU PACKAGE DES CONSTRAINTES ASSOCIEES AUX TRANSACTIONS                  *
*****************************************************************************************/
        -- modification au package des transactions TRANSACTIONS_CONS_PKG
        PACKAGE BODY TRANSACTIONS_CONS_PKG IS

        PROCEDURE CHECK_REGLE9 (CUSTID IN CLIENTS.CLI_NUMID%TYPE) IS
        nombre NUMBER;
        BEGIN
        SELECT COUNT(*) INTO nombre
                FROM TRANSACTIONS T, CLIENTS C
                WHERE T.TRA_CLI_NUMID = C.CLI_NUMID
                AND T.TRA_CLI_NUMID = CLIID
                AND C.CLI_TYPE='ETRANGER'
                GROUP BY T.TRA_CLI_NUMID;
                IF (nombre > 0) THEN
                RAISE_APPLICATION_ERROR (ERREURS.UP_TRA_REG9, ERREURS.UP_TRA_REG9_MSG)[3];
                ENDIF;
        END CHECK_REGLE9;

        FUNCTION CHECK_CON9 IS
        BEGIN
                -- DANS LE CAS D'UNE OPERATION DELETE CASCADE, LA CONTRAINTE EST IGNOREE
                PUISQUE LE FLAG EST ÉVALUÉ À TRUE
                IF CLIENTS_TAB_FLAG OR PROD_FINANCIER_TAB_FLAG THEN RETURN;
                END IF;
                -- UTILISATION DU PACKAGE CLI_ID_TABLE_PKG ASSOCIE A LA TABLE TEMPORAIRE
                 -- PL/SQL.
                CLI_ID_TABLE_PKG.SET_ITERATOR;
                WHILE CLI_ID_TABLE_PKG.MORE_IN_TABLE LOOP
                        CHECK_REGLE9 (CLI_ID_TABLE_PKG.NEXT_VAL);
                END LOOP;
        END CHECK_CON9;
```

On peut remarquer que la fonction CHECK_CON9 permet d'éviter la vérification de la contrainte en cas de suppression DELETE CASCADE sur la table TRANSACTIONS. En cas d'une suppression directe dans la table TRANSACTIONS, la règle doit être vérifiée par le trigger relatif

[3] Cette erreur doit être insérée au package ERREURS défini précédemment.

à une opération DELETE sur cette table. Nous ne l'avons pas écrit pour ne pas surcharger ce point.

2) Stratégie utilisant les exceptions ignorant les erreurs de type "table mutante".

L'écriture de cette partie est beaucoup plus simple puisqu'il suffit de déterminer les cas d'exception et de les ignorer. Il suffit de modifier la procédure du package TRANSACTIONS_CONS_PKG.CHECK_REGLE9 en déterminant les cas de génération d'erreurs de type "table mutante" et d'exiger d'ignorer ces erreurs; Il faut donc déclarer une exception (ERREUR_TABLE_MUTANTE) et lorsque cette exception est détectée, ignorer celle-ci et continuer l'opération (WHEN ERREUR_TABLE_MUTANTE THEN NULL).

7.2.2 Application de la règle dans tous les cas de DELETE CASCADE.

Quelles que soient les situations (DELETE CASCADE ou suppression directe dans la table TRANSACTIONS), la règle doit s'appliquer. Pour implémenter cette règle, un package associé à la suppression en cascade doit être créé. Il contiendra les différents flags relatifs aux tables "parent".

```
/*****************************************************************************
*                                ECRITURE DU PACKAGE DES FLAGS                *
*****************************************************************************/
        CREATE OR REPLACE PACKAGE DELETE_CASCADE_FLAG_PKG IS
                CLIENTS_TAB_FLAG BOOLEAN :=FALSE;
                PROD_FINANCIER_TAB_FLAG BOOLEAN :=FALSE;
        END DELETE_CASCADE_FLAG_PKG;
```

Le flag CLIENTS_TAB_FLAG est activé par un trigger before-delete et désactivé par trigger after-delete. Ces triggers s'occupent également de la gestion de la table temporaire. La table PL/SQL est remplie par un trigger associé à la table TRANSACTIONS. Dégageons la manière de modéliser cette règle par l'écriture des triggers associés à la table CLIENTS et TRANSACTIONS.

```
/*****************************************************************************
*              ECRITURE DES TRIGGERS ASSOCIES A LA TABLE "PARENT" CLIENTS     *
*****************************************************************************/
        CREATE OR REPLACE TRIGGER CLIENTS_BDS
        BEFORE DELETE ON CLIENTS
        BEGIN
                -- ACTIVER LE FLAG ET EFFACER LA TABLE TEMPORAIRE
                DELETE_CASCADE_FLAG_PKG.PROD_FINANCIER_TAB_FLAG:=TRUE;
                CLI_ID_TABLE_PKG.CLEAR;
        END;

        CREATE OR REPLACE TRIGGER CLIENTS_ADS
        AFTER DELETE ON CLIENTS
        BEGIN
        --VERIFICATION SI IL Y A UNE VIOLATION DE CONTRAINTE DE LA REGLE9
        TRANSACTIONS_CONS_PKG.CHECK_GLOBREGLE9;
        DELETE_CASCADE_FLAG_PKG.CLIENTS_TAB_FLAG:=FALSE;
        END;
/*****************************************************************************
*              ECRITURE DES TRIGGERS ASSOCIES A LA TABLE TRANSACTIONS         *
*****************************************************************************/
        CREATE OR REPLACE TRIGGER TRANSACTIONS_BDS
        BEFORE DELETE ON TRANSACTIONS
        BEGIN
                IF !CLIENTS_TAB_FLAG AND !PROD_FINANCIER_TAB_FLAG THEN
                        -- IL S'AGIT D'UNE SUPPRESSION SUR LA TABLE TRANSACTIONS
                        CLI_ID_TABLE_PKG.CLEAR;
                END IF;
```

```
END;

CREATE OR REPLACE TRIGGER TRANSACTIONS_ADR
AFTER DELETE ON TRANSACTIONS
AFTER EACH ROW
BEGIN
        -- ENREGISTRER LES VALEURS DE CORRELATION DANS LA TABLE PL:SQL
        CLI_ID_TABLE_PKG. PUT (:CLI_ID);
END;

CREATE OR REPLACE TRIGGER TRANSACTIONS_ADS
AFTER DELETE ON TRANSACTIONS
BEGIN
        IF !CLIENTS_TAB_FLAG AND !PROD_FINANCIER_TAB_FLAG THEN
                TRANSACTIONS_CONS_PKG.CHECK_GLOBREGLE9;
        END IF;
END;
```

L'exécution des triggers est complètement dépendante de la valeur des variables globales. Notons que, si une exception est générée, la transaction est annulée mais les flags ne sont pas réinitialisés. Pour résoudre ce problème, un processus du after-statement doit gérer, au cours du traitement de cette exception, la désactivation des flags.

Les triggers gérant les opérations de suppression DELETE CASCADE se multiplient en fonction du nombre de clés étrangères concernées et des règles à implémenter. Une personnalisation des suppressions en cascade peut générer une multitude de triggers de complexité croissante. Une attention particulière doit y être apportée afin de garantir la maintenance, la durabilité et une plus grande flexibilité des règles.

Conclusion

Conclusion

Apparu au milieu des années 1970, le domaine des bases de données actives connaît, depuis une dizaine d'années, un intérêt croissant de la part des chercheurs et des firmes commerciales. Si de nombreuses recherches ont été entreprises dans ce domaine afin de mieux comprendre leurs concepts et leurs techniques d'implémentation, les concepteurs des SGBD ont été convaincus de l'intérêt d'utiliser ces techniques afin d'accroître les fonctionnalités des bases de données.

Malgré ce potentiel, les développeurs ont très peu intégré ces mécanismes dans leurs applications. Généralement, le manque de connaissances de ces techniques et les méthodes d'implémentation relativement complexes ont été les principales causes de leur non-utilisation ou du moins de leur utilisation dans des fonctionnalités très simples. Face à cette méconnaissance, notre travail a tenté d'éclairer les développeurs et les utilisateurs sur le potentiel des règles actives.

Notre premier chapitre fournit une définition des systèmes de gestion de bases de données actives en dégageant leurs principales caractéristiques. En effet, un SGBDA est un système de gestion de bases de données implémentant le modèle ECA (Evénement-Condition-Action). Il doit supporter la gestion de ces règles en fonction de leur évolution, posséder également un modèle d'exécution, plusieurs modes d'association et de consommation, gérer un historique des occurrences et mettre en œuvre des techniques de résolution de conflits. Ce système relativement complexe apporte une nouvelle dynamique à la gestion des bases de données car il peut, indépendamment de l'intervention d'un utilisateur ou d'une application, réagir à certains événements en exécutant des actions spécifiées.

Ces interventions permettent d'apporter un contrôle plus approfondi, un renforcement de la sécurité, et une garantie de la cohérence de la base en fonction des règles stratégiques intégrées. Elles permettent, en outre, d'étendre les domaines d'application (le contrôle de processus industriels, la gestion de système d'informations, la sécurité de réseaux...), de renforcer la gestion des bases de données temporelles ou distribuées, et de fournir de l'information relative aux actions effectuées sur la base (statistique, audit...).

Selon les caractéristiques que nous avons dégagées, nous avons proposé une classification des systèmes actifs. Deux critères ont retenu notre attention et forment par ailleurs les principales variables de notre choix :

- le rôle (monitoring ou control) joué par le SGBDA dans le système d'information ;
- le degré d'intégration du système d'information (homogène ou hétérogène).

L'application des règles ECA se concentre principalement sur trois axes : la maintenance de l'intégrité (vérification et correction des contraintes), la gestion des vues et des transactions avancées et l'intégration des données dans les bases de données distribuées. Si les critères de cohérence et de sécurité des données sont les principaux objectifs des règles stratégiques, leurs utilisations explorent des domaines bien plus larges. La réflexion au second chapitre a étayé cette analyse.

Pour implémenter ces règles actives à travers le mécanisme des triggers et des procédures stockées, nous avons choisi le langage PL/SQL d'Oracle que nous avons brièvement décrit. PL/SQL est un langage de programmation procédurale, destiné au développement d'applications de bases de données. Il permet de combiner la logique procédurale avec le langage SQL.

Nous avons décrit en détails les procédures stockées PL/SQL qui sont des segments de codes procéduraux enregistrés directement dans la base de données. Oracle définit trois types de procédures stockées : les fonctions, les procédures et les packages. Les procédures et les fonctions sont des portions de code autonomes. Les packages encapsulent des procédures et des fonctions, des définitions de types et des déclarations de données. L'utilisation des procédures stockées permet d'assurer une certaine sécurité en réduisant les opérations pouvant être appliquées sur la base de données. Elle permet également d'améliorer la productivité des développements sans nuire à la performance de la base. Ce critère de performance est fondamental car l'utilisation abusive des procédures stockées et des triggers, associés à une méconnaissance des techniques d'implémentation, peut perturber gravement l'efficacité de la base de données.

Les triggers sont des procédures stockées exécutées implicitement par un SGBD en réponse à l'exécution d'une instruction d'insertion, de suppression ou de mise à jour. Le mécanisme des triggers implémente le modèle ECA. Il est possible de classifier les triggers selon trois typologies :

- en fonction de l'événement déclencheur (INSERT, UPDATE, DELETE);
- en fonction du nombre d'exécution du trigger (Statement or Row Level)
- en fonction du moment de déclenchement du trigger (Before or After).

Cette classification permet d'associer douze types de triggers à chaque table. Des conventions d'écriture ont été énoncées afin de mieux comprendre les éventuelles erreurs.

Le mécanisme des triggers est confronté à certaines restrictions d'utilisation. En effet, les opérations définies au sein du corps d'un trigger "Row Level" doivent respecter certaines limitations d'écriture (pas d'opération de lecture dans le corps du trigger sur la table définie comme cible de l'instruction SQL associée...). En outre, d'autres restrictions plus globales sont appliquées à l'usage des triggers : le langage associé au trigger possède généralement un caractère expressif limité, surtout dans la partie spécification d'évènements; le modèle d'exécution des triggers peut être trop restreint pour certaines applications... Grâce aux progrès des recherches en cette matière, ces limitations seront sûrement à l'avenir mieux intégrées.

Le modèle de bases de données actives doit intégrer conjointement les triggers procéduraux et le modèle des contraintes déclaratives. L'utilisation des contraintes déclaratives doit être privilégiée par rapport à la modélisation des triggers car elle est en passe de devenir un standard dans la déclaration des contraintes. Cependant, ce mécanisme ne suffit pas pour intégrer toutes les règles dans la base de données. Les triggers prennent alors le relais des contraintes déclaratives. Pour des règles devant subir de fréquentes modifications, la technique des triggers est également recommandée.

Le cas pratique que nous avons présenté, a mis en évidence le degré de difficulté de l'usage des triggers. Une méthodologie de développement a été présentée afin de préparer l'implémentation de règles stratégiques à insérer dans la base. L'identification des règles à

implémenter de façon procédurale constitue la première étape du processus. La construction d'une liste des violations de contraintes permet de dégager les évènements associés aux triggers et sur base de ce constat, de donner une description fonctionnelle du trigger. L'étape suivante définit les erreurs et les actions à opérer avant d'encapsuler les fonctionnalités au sein de packages. Une analyse des conflits éventuels est à examiner avant l'écriture des triggers qui devront encore être testés.

Au cours de ce développement, des conflits peuvent apparaître suite à des interférences venant d'un trigger lorsqu'une table est mutante (c'est-à-dire modifiée par le SGBD). Pour résoudre le conflit, il est nécessaire de recourir à une table temporaire qui enregistre certaines informations et de substituer certains triggers "Row-Level" en "Statement-Level". En outre, la gestion de la table temporaire exige l'écriture d'autres triggers. Par cette technique, les problèmes de conflits (table mutating) peuvent être surmontés.

Pour clôturer notre analyse, nous avons présenté certaines techniques de modélisation pour les triggers récursifs et pour les triggers intervenant dans des opérations de "DELETE CASCADE".

Dans la continuité de notre analyse, différentes recherches se concentrent actuellement sur l'amélioration des techniques afin d'améliorer les méthodes de modélisation, d'implémentation et de gestion des règles. En effet, les outils CASE tentent d'améliorer le support méthodologique de création des triggers, de fournir des mécanismes de gestion et de contrôle d'intégrité et d'introduire des dimensions graphiques afin d'assurer une gestion plus efficace des règles. Ils permettent également de surveiller la performance de la base de données par rapport à l'intégration des triggers. Voici quelques domaines de recherche qui pourraient s'insérer à la suite de notre analyse. Gageons que ces nouvelles techniques ne peuvent qu'apporter une nouvelle preuve de la dimension dynamique et stratégique des règles actives au sein des bases de données. L'intégration des triggers et des procédures stockées confirment l'évolution des systèmes de gestion de bases de données dans les aspects théoriques mais d'une manière plus tangible, dans les applications courantes.

Bibliographie

Bibliographie

Références bibliographiques

- Campin J., Paton N. W., and Williams M. H. (1995). *A Structured Specification of an Active Database System. Information and Software Technology*, 37(1):47-61.

- Ceri S. and Fraternali P." *Designing Database Applications with Objects and Rules*" Addison-Wesley, 1997.

- Ceri S. et Widom J., "*Managing Semantics Heterogeneity with production Rules and Persistent Queries*, 19th Intl Conference on Very Large Databases, P227, Springer, 1992.

- Ceri S. et Widom J., "*Deriving production Rules for incremental View Maintenace*, 17th Intl Conference on Very Large Databases, P577-589, Morgan Kaufmann, 1991.

- Chakravarthy S., Krishnaprasad V., Anwar E., and Kim S. K. (1994b*). Composite Events for Active Databases:Semantics Contexts and* Detection. In Proceedings of the 20th International Conference on Very Large Databases, pages 606-617.

- Chakravarthy S.: "*special issue on active databases*", Bulletin of the TC on data Engineering 15, 1992, 1-4.

- Cochrane R., Pirahesh H., Mattos N., "*Integrating Triggers and Declarative Constraints in SQL Database Systems*", In Proceedings of the 22th International Conference on Very Large Databases, Bombay, India, 1996.

- Dayal, Umeshwar, "*Active Database Systems; Triggers and Rules for Advanced Database Processing*", Morgan Kaufmann Publishers, 1995.

- Delmal Pierre, "*SQL2- application à Oracle, Access et RDB*", De Boeck Université, 1998.

- Dittrich K., Gatziu S. et Geppert A., *"The active database management system manisfesto: a rulebase of ADBMS features*", ACT-NET Consortium, 1997.

- Embury S. et Gray P., "*Database Internal Applications*", Monographs in Computer Sciences, springer, 1998.

- Fraternali P., Paraboschi S., Chimera*: A Language for designing Rule Applications*, Monographs in Computer Sciences, springer, 1998.

- Gehan N.i, Jagadish H.V., and Shumeli O., *"Advanced Database Systems, chapter Chapter 1:* COMPOSE*: A System For Composite Specification And Detection".* Springer LNCS 759, 1993.

- Gertz M., "*Specifying Reactive Integrity Control for Active Database*", in J. Widom and S. chakravarthy, editors, Proc. Of 4 th Intl Workshop on Reseach Issues in Data Engineering, Houston, Feb 94, P 42.

- Hainaut J-L, "*Conception et technologie des bases de données*", cours de 1ière licence, FUNDP, Namur, 1997.

- Hainaut J-L, "*Conception des bases de données : matières approfondies*", cours de 2ième licence, FUNDP, Namur,1998.

- Hanson E., "*The ariel project. In Active Database Systems - Triggers and Rules For Advanced Database Processing*", chapter 3, pages 63--86. Morgan Kaufman Publishers Inc., 1996.

- Knolmayer G., Herbst H. et *Schilesinger* M., *"Enforcing business Rules by applications of triggers Concepts*", Information Conference, Swiss National Science Foundation,1994.

- Kotz-Dittrich A. et Simon E., "*Active Database system, Expectations, Commercial Experience and beyond*" , Monographs in Computer Sciences, springer, 1998.

- Kulkarni K. Mattos N., Cochrane R., "*Active Datase features in SQL3*", Monographs in Computer Sciences, springer, 1998.

- Lieuwen D., Gehani N., and Arlein R, *"The Ode Active Database: Trigger Semantics and Implementation"*. In Proceedings of the 12th International Conference on Data Engineering, pages 412 - 420, 1996;

- Melton J.,"*Understanding SQL's Stored Procedures; A Complete* Guide to SQL/Psm", Morgan Kaufmann Publishers, 1998.

- Nicolas J-M., "*Logic for Improving Integrity Checking in Relational Databases"*, Acta Informatica, Vol 18, 227-240, 1992.

- Norman W. Paton, "*Active Rules in Database systems*", Monographs in Computer Sciences, springer, 1998.

- Oracle, "*Techniques for Tuning PL/SQL Applications*", An Oracle Technical White Paper, March 1999.

- Ostermann J-C., "*Oracle 7 - SQL, SQL plus, PL-SQL*", ENI, 1999.

- Owens, Kevin T.,"*Building Intelligent Databases with Oracle PL/SQL, Triggers, & Stored Procedures*", Prentice Hall, 1998.

- Rodgers, Ulka ; Rodgers ,"*Oracle; A Database Developer's Guide*", Prentice Hall, 1999.

- Sistla A. P. and Wolfson O., "*Temporal Triggers in Active Databases*", IEEE Transactions on Knowledge and Data Engineering, 7(3), pp. 471-486, 1995.

- Stonebraker M., Jhingran A., Goh J. et Potamianos, "*On Rules, Procedures, Caching and Views in DataBase Systems*", in H. Garcia-Molina and H.V. Jagadish, editors, Proc of SIGMOD 90, P.281-90, Atlantic City, May 1990.

- Urban S., Karadimce A et Nannapaneni R., "*Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database*, In 8th Intl. Conference on Data Engineering, p 565-572, Phoenix, Arizona, 1992. IEEE Computer Sciences.

- Toman and Chomicki, "*Implementing Temporal Integrity Constraints Using An Active* Database, in Widom and S. chakravarthy, editors, Proc. Of 4 th Intl Workshop on Reseach Issues in Data Engineering, Houston, Feb 94.

Référence Internet

Oracle, Triggers et Procedures Stockées

- http://www.oracle.com: aide en ligne des produits Oracle.

- http://www.oracle.com/oramag/ : Oracle magazines.

- http://education.oracle.com/: Oracle Education on line.

- http://www.orafans.com/ :The Oracle User Forum.

- http://www.onwe.co.za/frank/faq.htm : Oracle Frequendly Asked Questions.

- http://www.multimania.com/lewisdj/sql/chap15.htm : Triggers : application de l'intégrité référentielle.

- http://www.multimania.com/lewisdj/sql/chap14.htm : Utilisation des procédures stockées.

PL/SQL

- http://www-inf.int-evry.fr/~defude/COURS_BD/DOC_ORACLE/myplsql.html : PL/SQL.

- http://www.sfi-software.com/sql-debug-debugger.htm : SQL-Programmer with PL/SQL Debugger.

- http://cui.unige.ch/db-research/Enseignement/analyseinfo/PLSQL21/BNFindex.html : BNF Index of PL SQL version 2.1 for Oracle 7.

Les bases actives

- http://www.ida.his.se/ida/adc/ : The Active Database Central.

- http://www.loria.fr/~skaf/temporal.html : Temporal and Active DataBases.

- http://www-db.stanford.edu/classes.html : Database Classes and Seminars.

- http://www.elet.polimi.it/section/compeng/db/active/: Active Databases.

- http://www-db.research.bell-labs.com/user/hull/pubs-active.html: Papers on Active Databases.
- http://liinwww.ira.uka.de/bibliography/Database/adc.html: Bibliography on active databases.
- http://www.cee.hw.ac.uk/Databases/active.html: Analysis and Optimisation of Active Rules in Object-Oriented Databases.
- ftp://ftp.informatik.uni-hannover.de/papers/index.html: Papers of the Database and Information Systems Group.
- http://www.elet.polimi.it/users/dei/sections/compeng/stefano.ceri/methodology/methodology.htm: organization of the IDEA methodology.
- http://www.ifi.unizh.ch/groups/dbtg/SAMOS/samos.html: The SAMOS Project.

Annexe:

Les contraintes déclaratives

Les contraintes déclaratives

La connaissance des contraintes déclaratives est indispensable dans tout projet de création d'une base de données. Une bonne compréhension de leurs potentialités permet d'intégrer différents mécanismes de vérification de contraintes. Cette technique permet d'introduire un mécanisme de contraintes d'intégrité basées sur des règles établies. La sécurité, la stabilité, la fiabilité de la base en dépendent.

Les développeurs de SGBD au moment de la création de la base ont de plus en plus tendance à renforcer les mécanismes de lutte contre la violation des contraintes. Les règles émises doivent obligatoirement être respectées à tout moment et en tout lieu (en cas de bases distribuées). La syntaxe des contraintes déclaratives n'est pas le facteur le plus critique puisqu'une abondante littérature fournit les principes d'écriture. Cependant, le facteur le plus important est de pouvoir évaluer les différents coûts des alternatives (contraintes déclaratives et autres options) qui existent afin d'introduire ces règles d'intégrité.
Une utilisation adéquate des contraintes déclaratives permet de réduire le code de l'application. Il est cependant complètement injustifié d'insérer des règles dans le code de l'application lorsque les contraintes déclaratives peuvent remplir ces mêmes fonctions d'une manière plus efficace.

Nous allons décrire les différents types de contraintes procédurales permettant d'assurer l'intégrité des données par rapport aux règles émises.

Les références de notre analyse sont :

- L'aide en ligne d'Oracle fournie avec le logiciel.
- http://www.oracle.com.
- Pierre Delmal, "*SQL2 - application à Oracle, Access et RDB*", De Boeck Université, 1998.
- Jean-Claude Ostermann, "*Oracle 7 - SQL, SQL plus, PL-SQL*", ENI, 1999.
- Rodgers, Ulka; Rodgers ,"*Oracle; A Database Developer's Guid*e", Prentice Hall, 1999.
- Owens, Kevin T.,"*Building Intelligent Databases with Oracle PL/SQL, Triggers, & Stored Procedures*", Prentice Hall, 1998.

Sachons, cependant, que le mécanisme des triggers permet, de la même manière, d'intégrer des contraintes procédurales au sein d'une base de données mais d'une manière beaucoup plus puissante. Nous allons commenter les principales contraintes déclaratives telle qu'elles peuvent être définies au sein des SGBD classiques Nous aborderons les cinq contraintes déclaratives classiques présentes dans le plupart des systèmes de gestion de base de données :

- La clé primaire
- Contrainte de type UNIQUE
- Contrainte de type NOT NULL
- Contrainte de type CHECK
- La clé étrangère

I) La clé primaire (PRIMARY KEY)

La clé primaire impose que la valeur d'une colonne ou d'un ensemble de colonnes d'une table donnée soit **unique et obligatoire**. Même si une table ne doit pas posséder obligatoirement une clé primaire, il est cependant fortement recommandé que chaque table en possède une.
Une table peut avoir au plus une clé primaire fixée sur une ou plusieurs colonnes. L'existence d'une clé primaire est, dès lors, la garantie que la valeur associée à la clé primaire identifie un et un seul enregistrement de la table concernée. Conjointement, la création d'une clé primaire établit un index qui permet d'optimiser les exécutions d'instructions de recherche (SELECT).

Il existe plusieurs alternatives pour déclarer une clé primaire sur une table.

Il est également possible de définir la contrainte en intégrant la contrainte via l'instruction CREATE TABLE ou de préférence, par l'instruction ALTER TABLE. Les exemples suivants nous montrent les différentes techniques de déclaration d'une clé primaire. Il est recommandé de donner un nom à la contrainte de clé primaire plutôt que de laisser le SGDB en générer un.

Supposons une table Clients composée des champs CLI_NUMID, CLI_NOM, CLI_PRENOM représentant respectivement le numéro d'un client, son nom et son prénom. Le premier champ CLI_NUMID est déclaré comme une clé primaire. Voyons la déclaration de cette table selon les différentes alternatives.

1) Cette commande utilise une clause de contrainte associée à une colonne. Le nom de la contrainte PK_CLIENTS permet d'associer directement le nom de la contrainte à sa fonction.

```
CREATE TABLE CLIENTS (
CLI_NUMID NUMBER(7) CONSTRAINT PK_CLIENTS PRIMARY KEY ,
CLI_NOM VARCHAR(50) ,
CLI_PRENOM VARCHAR(30) ,
...) ;
```

2) Cette commande utilise une clause de contrainte de table.

```
CREATE TABLE CLIENTS (
CLI_NUMID NUMBER(7)
CLI_NOM VARCHAR(50) ,
CLI_PRENOM VARCHAR(30) ,
...,
CONSTRAINT PK_CLIENTS PRIMARY KEY (CLI_NUMID)) ;
```

3) La commande suivante crée une table Clients dont le champs CLI_NUMID est déclarée NOT NULL (champ obligatoire). Conjointement, la commande ALTER TABLE déclare la clé primaire et génère un index pour celle-ci.

```
CREATE TABLE CLIENTS (
CLI_NUMID NUMBER(7) NOT NULL
CLI_NOM VARCHAR(50) ,
CLI_PRENOM VARCHAR(30) ,
TABLESPACE CLIENTS_TS) ;
ALTER TABLE CLIENTS ADD CONTRAINT
PK_CLIENTS PRIMARY KEY (CLI_NUMID) TABLESPACE client_idx_ts
```

Le troisième exemple présente quelques avantages par rapport aux autres alternatives : la contrainte de clé primaire déclarée par la commande ALTER TABLE permet de créer la table des données et l'index associé dans des espaces dissociés. Comme dans les trois alternatives, la contrainte générée est identifiée nominalement; ce qui permet de détecter plus facilement

une violation associée. Enfin, dans la troisième alternative, le champ clé primaire est déclaré par une contrainte additionnelle NOT NULL qui peut être mise en évidence par les commandes SQL*PLUS de description de table DESCRIBE[1].

Remarques :

Lorsqu'il est difficile de définir une clé primaire sur une table, quelle qu'en soit la raison, il est fréquent de créer une clé primaire technique pour résoudre ce type d'inconvénient; il s'agit d'un champ qui est défini uniquement pour des raisons techniques dont la maîtrise est généralement déléguée au SGBD (incrémentation automatique).

La définition d'une contrainte PRIMARY KEY ou UNIQUE (cfr point suivant) peut se définir sur plusieurs colonnes. Il s'agit dans ce cas d'une clé primaire concaténée (CONCATENATED PRIMARY KEY). Cependant, si une ou plusieurs colonnes de la clé primaire peut être définies de type NULL, si tous les champs de la clé concaténée sont vides, alors une erreur sera générée. Supposons la table TRANSACTIONS dont la clé primaire (clé concaténée) est définie sur deux champs TRA_TYPE (le type de transactions) et TRA_NUM (le numéro de la transaction d'un type donné) :

```
CREATE TABLE TRANSACTIONS (
TRA_TYPE VARCHAR(15) ,
TRA_NUM NUMBER(7),
...) ;

ALTER TABLE TRANSACTIONS ADD CONTRAINT
PK_TRANSACTIONS PRIMARY KEY (TRA_TYPE, TRA_NUM) ;
```

(définition de la clé primaire concaténée)

Si les données encodées sont les suivantes, regardons la validité de l'encodage

TRA_TYPE	TRA_NUM	Validation des données
SWAP	0126790	← OK
Null	0123456	← OK
OPTION	2323445	← OK
Null	Null	← Erreur générée car toutes les colonnes de la clé primaire sont vides.

Lorsqu'une instruction de mise à jour ou d'insertion viole la contrainte de clé primaire, plusieurs messages d'erreurs peuvent être générés. Donnons, à titre documentaire, certains messages générés par Oracle en réponse à la violation de certaines contraintes de clé primaire.

- Lorsque une instruction ajoute un enregistrement ou met à jour un enregistrement où l'identifiant est déjà présent dans la table, le message de violation généré sera : ORA-00001 : unique constraint (DUPONT[2]PK_CLIENTS) violated

[1] La commande DESCRIBE permet de présenter une table à travers les noms des colonnes, leur type de données ainsi que les contraintes NOT NULL associées. La commande DESC (raccourci pour la commande DESCRIBE) CLIENTS sur le troisième exemple nous présenterait le tableau suivant :

NAME	NULL ?	TYPE
CLI_NUMID	NOT NULL	NUMBER(7)
CLI_NOM		VARCHAR(50)
CLI_PRENOM		VARCHAR(30)

[2] Représentation du nom de l'utilisateur actuel.

- Si une commande d'insertion ajoute une enregistrement dont la clé primaire est vide (NULL) : ORA-01400 : mandatory (NOT NULL) column is missing or null during insert
- Si une opération UPDATE met à jour un enregistrement en supprimant la clé primaire. ORA 01407 : cannot update (NOT NULL) column to NULL
- Lorsqu'une contrainte n'est pas nommée explicitement, le SGBD génère, lui-même, un nom de contrainte dont la structure en Oracle est une chaîne de caractères commençant par« SYS_C » suivi d'une série de six chiffres. Cette nomination est peu expressive; ce qui peut rendre plus complexe la recherche de la table concernée par rapport au message d'erreur affiché. ORA-000001: unique constraint (DUPONT.SYS_C012343) violated. Il est important pour la lisibilité des messages d'erreurs de nommer les contraintes et également de différencier les messages en fonction du type d'erreur associée[3]. Les temps d'analyse et de correction des erreurs en seront optimisés.

II) CONTRAINTE DE TYPE UNIQUE

La contrainte UNIQUE est équivalente à la contrainte de clé primaire à l'exception près que la valeur NULL est permise. Il est important de ne pas confondre la valeur NULL et la valeur nulle (zéro). La valeur NULL représente l'absence de données pour un champ; ce qui n'a aucune comparaison avec la valeur nulle.

L'avantage de ce type de contrainte est que, si une table ne peut avoir qu'une seule clé primaire, elle peut posséder plusieurs champs de type UNIQUE. Les colonnes déclarées UNIQUE possèdent des attributs qui sont à la fois facultatifs et singuliers (s'ils existent). Une clé étrangère peut donc faire référence à ce type de champs; tout comme à une clé primaire .

Par exemple, un client qui possède un numéro de compte, peut également avoir, facultativement, un numéro de référence vers un broker. Ce numéro, s'il existe, est obligatoirement unique. La création d'une contrainte UNIQUE engendre systématiquement un index sur la colonne.

Le script suivant permet de créer cette contrainte de type UNIQUE selon deux méthodes :
1) par une contrainte de colonne :

```
CREATE TABLE CLIENTS (
CLI_NUMID NUMBER(7)  CONSTRAINT PK_CLIENTS PRIMARY KEY ,
CLI_NOM VARCHAR(50) ,
CLI_PRENOM VARCHAR(30) ,
CLI_NUMBROKER NUMBER(7) CONSTRAINT UK_CLIENTS_NUMBROKER  UNIQUE,
...) ;
```

il est également d'associer cette contrainte avec une contrainte de table.

2) par une contrainte supplémentaire :

```
CREATE TABLE CLIENTS (
CLI_NUMID NUMBER(7), NOT NULL,
CLI_NOM VARCHAR(50) ,
```

[3] En cas de simulation d'une clé primaire, les messages d'erreurs peuvent être personnalisés en fonction de certains cas particuliers. Citons les messages d'erreurs en fonction des résultats des fonctions NO_DATA_FOUND OU TOO_MANY_ROWS.

```
CLI_PRENOM VARCHAR(30) ,
CLI_NUMBROKER NUMBER(7),
...) ;
ALTER TABLE CLIENTS ADD CONTRAINT
PK_CLIENTS (CLI_NUMID) PRIMARY KEY ;
ALTER TABLE CLIENTS ADD CONTRAINT
CONSTRAINT UK_CLIENTS_NUMBROKER (NUMBROKER) UNIQUE ;
```

Nous constatons que dans les deux cas, la contrainte est nommée suivant la convention : Nom_Contrainte_unique = UK[4]_NOMTABLE_NOMCOLONNE avec un maximum de 30 lettres.

En cas d'insertion ou de mise à jour d'un champ déjà inséré, le système générera une erreur qui est similaire à la même opération associée à une clé primaire :

ORA-00001 : unique constraint (DUPONT.UK.CLIENTS_NUMBROKER) violated.

Il est également possible de créer des contraintes UNIQUE sur un ensemble de champs :

```
ALTER TABLE CLIENTS ADD CONTRAINT
UK_CLIENTS_NUMBROKER (NUMBROKER, NOMBROKER) UNIQUE
```

où NUMBROKER et NOMBROKER représentent un champ concaténé de type UNIQUE. Ce qui signifie que plus d'un client ne peut avoir le même numéro et le même nom de broker. Un client ne négociant pas avec un broker doit avoir les deux champs NUMBROKER et NOMBROKER NULL ; cependant il est acceptable qu'un des deux champs soit NULL. Cette situation quelque peu illogique est relativement préoccupante. Cependant, une contrainte supplémentaire de type CHECK[5] peut éviter ces circonstances.

III) CONTRAINTE DE TYPE NOT NULL

Une contrainte NOT NULL garantit que la colonne concernée possède obligatoirement une valeur. Une colonne NOT NULL ne peut donc avoir des valeurs manquantes. Au contraire, le type NULL est le mot clé permettant d'indiquer la possibilité d'absence de valeurs dans le champ spécifié.

Ce type de contraintes permet d'assurer l'intégrité relative à la présence ou non d'une donnée. Si nous regardons la table Clients, il est raisonnable de définir, à coté de la clé primaire CLI_NUM, des champs obligatoires tels que le nom et l'adresse d'un client. En revanche, des champs comme le(la) conjoint(e) et le nombre d'enfants d'un client peuvent être facultatifs. Il est relativement clair que les contraintes de ce type sont essentielles pour garantir l'intégrité de la base. Imaginons le cas où lors d'une insertion d'une transaction, si le prix d'achat d'une option n'était pas encodé, cette situation causerait d'énormes problèmes lors du traitement comptable. Ce type de champ doit être déclaré NOT NULL. La déclaration de ces contraintes s'imposent généralement au bon sens.

La définition d'une contrainte NOT NULL au sein d'une déclaration de table s'établit selon les conventions suivantes :

```
CREATE TABLE TRANSACTIONS (
TRA_NUMID NUMBER(7) CONSTRAINT PK_TRANSACTIONS PRIMARY KEY ,
```

[4] UK pour UNIQUE KEY
[5] Voir le point IV.

```
...
TRA_TYPE VARCHAR(20) CONSTRAINT NN6_TRANSACTIONS_TYPE NOT NULL ,
TRA_PRIX NUMBER(10), CONSTRAINT NN_TRANSACTIONS_PRIX NOT NULL ,
TRA_QUANTITE NUMBER(10) , CONSTRAINT NN_TRANSACTIONS_QUANTITE NOT
NULL ,
...,
) ;
```

Si une opération d'insertion ou de mise à jour modifie un champ NOT NULL en le rendant vide, une erreur sera automatiquement soulevée.

En cas d'insertion : ORA-1400 : mandatory (NOT NULL) column is missing or NULL during insert.

En cas de mise à jour : ORA-1407 : update mandatory (NOT NULL) column to NULL.

Le nom de la contrainte NOT NULL n'apparaît pas dans l'exception soulevée en réponse à la violation de la contrainte. Il est généralement préféré de ne pas nommer ce genre de contraintes pour éviter la confusion avec les contraintes d'autres types. En outre, les contraintes NOT NULL sont les seuls types de contraintes dont le nom est généré par Oracle ; ce qui permet de les identifier plus facilement si l'on examine les informations sur les contraintes dans le dictionnaire.

Les valeurs d'une colonne peuvent être obligatoires dans certaines circonstances et facultatives dans d'autres cas. Cette situation ne peut être spécifiée avec une contrainte NOT NULL. Imaginons le cas où si le type d'opérations est « action », le champs porteur est facultatif ; par contre si le type d'opération est « obligation », le champs porteur est obligatoire. Si une contrainte NOT NULL est associée au champ porteur, la première situation ne peut s'effectuer. En revanche, si le champ n'a pas de contrainte, la champ porteur dans le cas d'un type d'obligation peut être vide ; ce qui est illicite par rapport aux règles spécifiées. Nous verrons plus tard le moyen de modéliser ce genre de contrainte.

IV) Les contraintes de type CHECK

Les contraintes de type CHECK permettent de délimiter les valeurs introduites dans une ou plusieurs colonnes. En effet, la liste des valeurs introduites peut être explicitement contrainte par rapport à une liste nommée (le type des transactions est action, obligation, swap, sicav.. et call), ou par rapport à une expression mathématique (l'âge du client est compris entre 18 et 125 ans). Ce type de contrainte permet également de personnaliser le type de contrainte booléenne (on - off ; 0 – 1, oui – non). Ce type de contrainte est très utilisé pour modéliser des règles d'intégrité plus complexes.

La définition des deux premières contraintes peut s'écrire dans la déclaration de la table comme suit :

```
TRA_TYPE VARCHAR(20) CONSTRAINT CHECK (TRA_TYPE IN ('action', 'obligation',
'swap', .., 'call')),
CLI_AGE NUMBER(3) CONSTRAINT CHECK (CLI_AGE BETWEEN 18 AND 125),
```

[6] Convention d'écriture NN pour NOT NULL

Remarques :

- Il est possible de combiner ce type de contrainte avec les contraintes précédemment analysées. La contrainte suivante permet de définir une colonne NOT NULL et de délimiter les valeurs d'encodage aux valeurs spécifiées.

  TRA_TYPE VARCHAR(20) **NOT NULL CONSTRAINT CHECK (TRA_TYPE IN ('action', 'obligation', 'swap', .., 'call')),**

- La vérification des contraintes de type CHECK respecte l'écriture des majuscules et minuscules (case sensitive). L'insertion d'une transaction de type "ACTION" générera automatiquement une erreur puisque seul le nom admissible est "action".
- Cependant, certaines fonctions permettent d'éviter ce type d'erreur. En effet, les fonctions INITCAP, LOWER et UPPER permettent respectivement de transformer la première lettre d'un mot en majuscules et de retourner l'expression en minuscules ou majuscules. Si un utilisateur tente d'insérer une opération dont le type TR_TYPE_VALEUR est introduit sans tenir compte des majuscules ou minuscules, l'opération INSERT INTO TRANSACTIONS (TRA_TYPE) VALUES (LOWER(TR_TYPE_VALEUR)) permettra d'assurer un encodage en minuscules. En effet, quelle que soit l'expression introduite, le système la transformera entièrement en minuscules.

- Il est possible de construire des règles plus complexes portant sur plusieurs colonnes. Cette situation permet de modéliser des contraintes beaucoup plus rigoureuses et complètes. Par exemple, toute transaction doit obligatoirement avoir un prix, une quantité et doit provenir d'un employé, soit du front office, soit du back office. Voici la déclaration de la contrainte permettant de modélisant cette règle.

  CONSTRAINT CH_ENCODAGE_TRANSACTIONS CHECK ((TRA_PRIX >= 0) AND (TRA_QUANTITE >= 0) AND (NOM_ENCODEUR IN ('back office', 'front office'));

- Nous avons vu comment la logique du CHECK peut être remplacée par la technique des triggers afin de modéliser des contraintes encore plus complexes. En effet, il peut arriver que des contraintes ne puissent être modélisées par le mécanisme des contraintes de type CHECK. La technique des triggers peut être analysée comme un substitut mais également comme un complément aux contraintes de type CHECK.

V) Les contraintes de clé étrangère (FOREIGN KEY)

- On appelle clé étrangère ou colonne de référence, toute colonne qui est constituée de l'identifiant d'une autre table et qui joue un rôle de référence à une ligne de cette table. Les valeurs qui forment la contrainte font partie de la colonne "parent" de la table "parent". Les valeurs qui sont contraintes s'intègrent dans la colonne "enfant" de la table "enfant". Cette relation de clé étrangère est associée à une relation "parent-enfant". Une contrainte de clé étrangère doit faire référence à une colonne qui, elle-même, possède une contrainte "PRIMARY KEY" ou "UNIQUE".

- Supposons une table "parent" qui enregistre toutes les informations relatives aux clients et une autre table qui stocke les transactions opérées par ces différents clients. En particulier, un champ TRA_NUM_CLI désigné comme clé étrangère, spécifie le numéro client qui a effectué la transaction. Le champ CLI_NUMID de la table clients doit être déclaré comme

clé primaire ou "UNIQUE" pour pouvoir déclarer le champ TRA_NUM_CLI comme clé étrangère par rapport à ce champ.

Pour garantir l'intégrité référentielle, différentes règles s'imposent :

Opérations effectuées	**Règle**	**Restrictions sur l'opération**
Insertion dans la table "enfant"	*CHILD-INSERT-RESTRICT*	On ne peut pas insérer une transaction dans la table TRANSACTIONS si le numéro de référence au client n'est pas défini dans la table CLIENTS.
Mise à jour dans la table "enfant"	*CHILD-UPDATE-RESTRICT*	On ne peut pas mettre à jour, au sein d'une transaction de la table TRANSACTIONS, un enregistrement si son numéro de référence de client qui est modifié, n'est pas défini dans la table CLIENTS.
Mise à jour dans la table "parent"	*PARENT-UPDATE-RESTRICT*	On ne peut pas mettre à jour le numéro de référence d'un client de la table CLIENTS en laissant des transactions dans la table TRANSACTIONS associées à aucun client existant.
Suppression dans la table "parent"	*PARENT-DELETE-RESTRICT*	On ne peut pas supprimer un enregistrement de la table CLIENTS en lui laissant des transactions associées.

Ces différentes règles sont cruciales lors de la création des triggers. Pour pouvoir vérifier ces différentes règles lors d'instructions d'insertion, de suppression ou de mise à jour, le SGBD s'assure, à travers plusieurs lectures, que les règles sont respectées avant d'exécuter ces instructions.

Si une règle est violée, une erreur sera directement générée. L'erreur suivante " ORA-02292: integrity constraint (DUPONT.TRA_NUM_CLI) violated - parent key not found" est générée en cas d'insertion ou de mise à jour d'un enregistrement dont le champ TRA_NUM_CLI n'est pas un numéro de client valide c'est-à-dire défini dans la table clients.

En outre, des clients peuvent exister dans la table CLIENTS indépendamment de toutes transactions. En effet, dans une relation "parent-enfant", un parent peut exister sans avoir des enfants. De plus, si une colonne associée à une clé étrangère est aussi déclarée avec une contrainte NOT NULL, le champ ainsi défini clé étrangère devient obligatoire. La création de la table TRANSACTIONS peut alors se définir comme suit :

```
CREATE TABLE TRANSACTIONS (
    TRA_NUMID NUMBER(7)  PRIMARY KEY ,
    TRA_NUM_CLI NUMBER(7) NOT NULL
    ...,
    CONSTRAINT FK_TRANSACTIONS_TRA_NUMID[7] FOREIGN KEY (TRA_NUM_CLI)
    REFERENCES CLIENTS (CLI_NUMID) );
```

Pour limiter les problèmes associés à la suppression d'un enregistrement "parent" qui possède encore des enregistrements "enfants", l'option "DELETE CASCADE" permet de manière

[7] Le nom de la contrainte respecte la convention FK_NOMTABLE_NOMCOLONNE; FK pour Foreign Key.

récursive de supprimer automatiquement tous les champs "enfant" associés à la suppression d'un enregistrement "parent".

Lorsqu'une contrainte de clé étrangère est définie avec l'option "DELETE CASCADE", l'objectif de cette option peut se définir comme suit : lorsqu'un enregistrement "parent" est supprimé, tous les enregistrements qui y font référence sont également supprimés. Par la suite, cette règle s'applique également aux enfants qui sont effacés et qui, à leur tour, sont des parents relatifs à d'autres enfants (grand-enfant). Cette option peut donc avoir des effets très importants sur une base de données et doit donc être utilisée de manière prudente pour ne pas effacer des enregistrements indispensables à d'autres relations.

La syntaxe d'une contrainte de clé étrangère peut être définie directement à la création d'une table de cette manière :

> Nom_colonne Type_donnée [CONSTRAINT Nom_contrainte] FOREIGN KEY Nom_colonne
> [,Nom_colonne,...]) REFERENCES Table [(Colonne[,Colonne..])] [ON DELETE CASCADE]

Le nom de la contrainte est optionnel. En l'absence de la partie CONSTRAINT, Oracle génère un nom de contrainte selon les conventions "SYS_C" suivi d'une série de six chiffres. En réalité, il est préférable de nommer cette contrainte pour des raisons de lisibilité des messages d'erreurs. Conventionnellement, le nom de la contrainte comporte la suite FK_ suivi du nom de la table et de la colonne concernée.

La partie FOREIGN KEY Nom_colonne est la colonne déclarée "clé étrangère". La clé étrangère peut porter sur plusieurs champs qui doivent obligatoirement faire référence à des champs concaténés définis avec une contrainte PRIMARY KEY ou UNIQUE dans une et une seule table.

La suite de l'instruction REFERENCES permet d'identifier la table "parent" impliquée et de préciser le ou les champs associés. Si ce ou ces champs ont été déclarés "clé primaire", il n'est pas nécessaire de les mentionner. Par défaut, la clause effectue la liaison avec la clé primaire de la table mentionnée.

L'option [ON DELETE CASCADE] est une caractéristique très utilisée en matière d'intégrité référentielle. Pour rappel, elle permet de supprimer tous les enregistrements "descendants" associés à la suppression d'un enregistrement "parent".

Il est possible que la référence d'une clé étrangère soit associée à une colonne clé primaire ou UNIQUE déclarée au sein de la même table. Une table Employé où tout employé dépend d'un directeur qui est lui-même un employé, en est un exemple.

VI) Remarques générales sur les contraintes déclaratives

En Oracle, les contraintes déclaratives à l'exception des contraintes NOT NULL, peuvent être déclarées temporairement inactives. En effet, l'option [DISABLE, ENABLE] associée à une contrainte permet de spécifier si la contrainte est effective dans l'application.

> CONSTRAINT FK_TRANSACTIONS_TRA_NUMID FOREIGN KEY (TRA_NUM_CLI) REFERENCES CLIENTS (CLI_NUMID) **DISABLE**);

- Il est possible d'associer une valeur par défaut à une colonne lors de la création d'une table. Par exemple, si une insertion d'un client ne précise pas le numéro du broker, la

valeur par défaut du champs doit être 0000010. Cette technique permet d'éviter que certains champs ne soient vides lors d'insertions.

```
CREATE TABLE CLIENTS (
CLI_NUMID NUMBER(7), NOT NULL,
CLI_NOM VARCHAR(50) ,
CLI_PRENOM VARCHAR(30) ,
CLI_NUMBROKER NUMBER(7) DEFAULT 0000010);
```

Les commandes ALTER TABLE permettent de modifier, sous certaines conditions, la définition de certaines colonnes, d'ajouter certains colonnes et de rendre opérationnelles certaines contraintes en activant le statut ENABLE de la contrainte concernée.